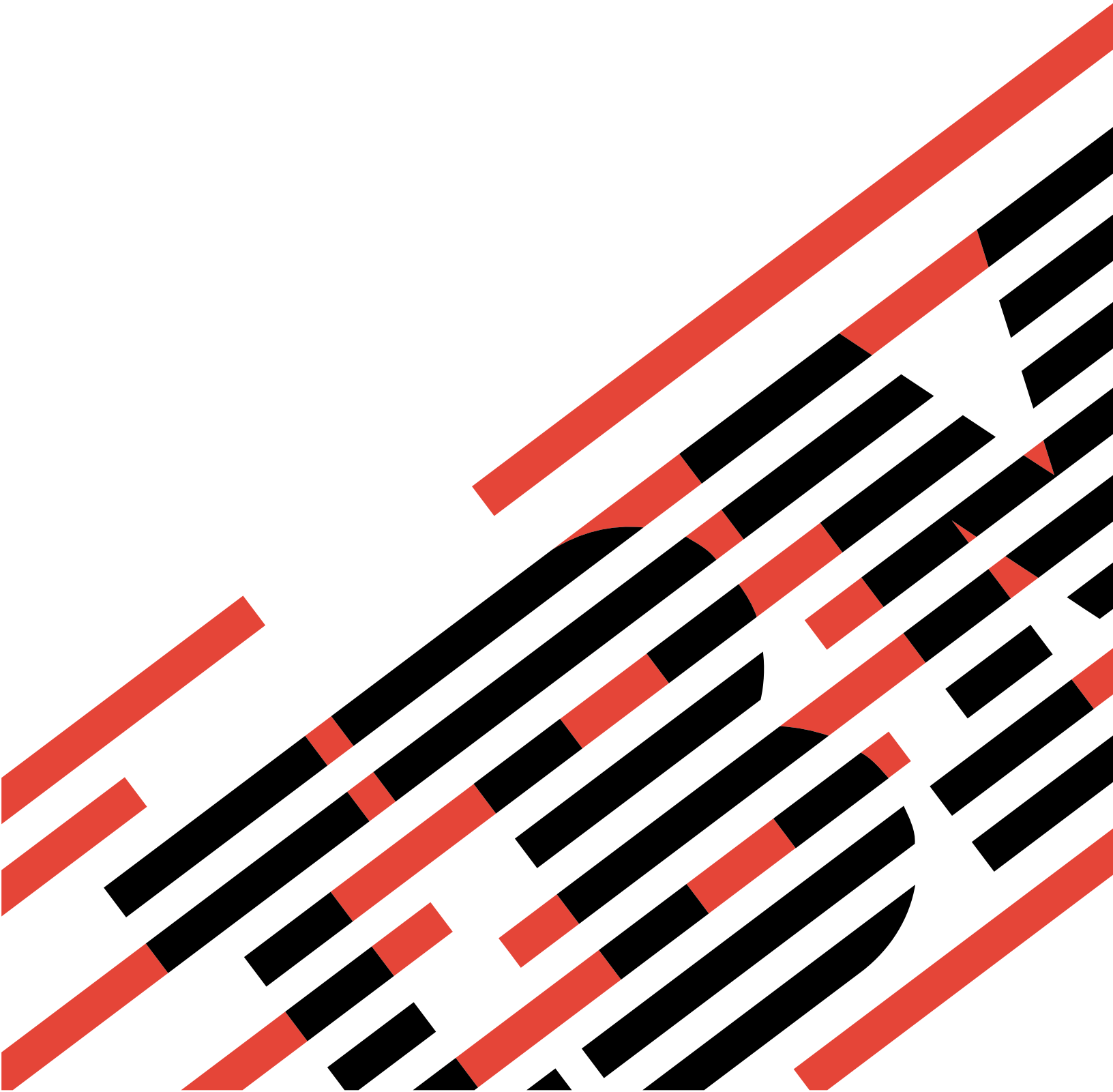
IBM

eserver

iSeries

소켓 프로그래밍



IBM

@server

iSeries

소켓 프로그래밍

목차

제 1 부 소켓 프로그래밍

소켓은 네트워크에서 이름과 주소를 지정할 수 있는 통신 연결 지점(종료점)입니다. 소켓을 사용하는 프로세스는 동일한 시스템에 있을 수도 있고 서로 다른 네트워크상의 다른 시스템에 있을 수도 있습니다. 소켓은 독립형 어플리케이션과 네트워크 어플리케이션 모두에 유용합니다. 소켓을 사용하여 동일한 기계에서 또는 네트워크를 통해 프로세스 사이에 정보를 교환하고 가장 효율적인 기계에 작업을 분배하며 중앙 집중된 자료에 쉽게 액세스할 수 있습니다. 소켓 API(어플리케이션 프로그램 인터페이스)는 TCP/IP의 네트워크 표준입니다. 광범위한 오퍼레이팅 시스템에서는 소켓 API를 지원합니다. OS/400 소켓은 복수 전송 및 네트워킹 프로토콜을 지원합니다. 소켓 시스템 함수와 소켓 네트워크 함수는 스레드세이프 상태에 있습니다.

소켓 프로그래밍은 리모트 및 로컬 프로세스간 통신 링크를 설정하는 데 소켓 어플리케이션 프로그램 인터페이스(API)를사용하는 방법을 보여줍니다. 통합 언어 환경(ILE) C를 사용하는 프로그래머는 소켓 어플리케이션을 개발할 정보를 사용할 수 있습니다. 또한, RPG와 같은 기타 ILE 언어에서 소켓 API로 코딩할 수도 있습니다. ILE RPG에 대한 자세한 정보는 IBM 레드북 Who Knew You Could Do That with RPG IV? A Sorcerer's Guide to System Access and More  를 참조하십시오.

Java에서는 소켓 프로그래밍 인터페이스도 지원합니다. 자세한 내용은 Information Center의 Java 주제를 참조하십시오.

소켓 프로그래밍 주제:

다음 주제에는 소켓 어플리케이션을 개발하는 데 도움이 되는 개념, 설계 권장사항 및 예가 나와 있습니다. 다음을 참조하십시오.

- **V5R2의 새로운 사항**
  소켓 프로그래밍과 관련된 새로운 기능을 학습하려면 이 페이지를 사용하십시오. 이 주제에는 이런 확장 기능에 대한 추가 정보를 제공하는 다른 페이지로의 링크가 있습니다.
- **이 주제 인쇄**
  소켓 프로그래밍 정보의 PDF(Portable Document Format) 버전을 인쇄하거나 다운로드하려면 이 페이지를 사용하십시오.
- **소켓 프로그래밍의 전제조건**
  이 주제에서는 소켓 API를 사용하여 어플리케이션을 작성하기 전에 완료해야 할 필수 타스크를 설명합니다.
- **기본 소켓 설계**
  이 주제에는 가장 기본적인 소켓 유형의 프로그램 예에 대한 개요가 나와 있습니다. 기본 소켓 설계 전략을 보여주는 예를 보려면 샘플 프로그램에 대한 링크를 사용하십시오.
- **소켓 개념**
  이 주제에서는 비동기 입/출력(I/O) 및 글로벌 보안 툴킷(GSKit)과 같은 보다 첨단 소켓 개념을 설명합니다. 개념과 연관된 샘플 프로그램을 검토하려면 이 주제의 링크를 사용하십시오.

- **소켓 시나리오: IPv4 및 IPv6 클라이언트를 허용하기 위한 어플리케이션 작성**
  이 주제에서는 AF_INET6 주소 패밀리를 사용할 일반적인 상황을 설명합니다. V5R2에서는 이 주소 패밀리가 인터넷 프로토콜 버전 6(IPv6)을 지원합니다. IPv6은 128비트 IP 주소를 지원합니다. 이 주제에는 AF_INET6 주소 패밀리를 사용하는 소켓 어플리케이션을 구현하는 데 사용할 수 있는 프로그램 예로의 링크 및 계획 정보도 있습니다.
- **소켓 어플리케이션 설계 권장사항**
  이 주제에는 보다 효율적인 소켓 어플리케이션 설계를 위한 힌트가 나와 있습니다.
- **예: 소켓 어플리케이션 설계**
  이 주제에는 소켓 어플리케이션을 작성하는 데 사용할 수 있는 소켓 프로그램 예가 나와 있습니다.

  주: 이 정보에는 코드 예가 있습니다. 이런 샘플 프로그램 사용에 대한 세부사항은 코드 면책사항 정보를 참조하십시오.
- **Xsocket 툴**
  이 주제에는 소켓 프로그래머가 소켓 어플리케이션 개발에 도움을 주는 데 사용할 수 있는 Xsocket 툴 설명이 나와 있습니다. 툴 설치 및 사용에 대한 지침을 읽으려면 이 주제의 링크를 사용하십시오.
- **서비스 능력 확장 툴**
  이 주제에는 소켓용 서비스 능력 확장 툴의 설명이 나와 있습니다.
- **관련된 정보**
  이 주제에서는 기타 소켓 정보에 대한 설명과 링크를 제공합니다.

제 1 장 V5R2의 새로운 사항

이 주제에서는 iSeries 소켓에 대한 기능적 확장 기능을 강조합니다. 소켓 프로그래밍 정보에서는 기본 및 확장 개념 둘 모두를 포함하고 있으며 샘플 프로그램 및 설계 권장사항도 제공됩니다. 또한, 소켓 프로그래밍 주제는 고급 및 초보 소켓 프로그래머 둘 모두를 위해 재구성되었습니다. 특정 프로그램 예를 사용하는 것에 관해 어플리케이션 프로그래머에게 스펙을 제공하는 시나리오도 포함되어 있습니다. 다음 리스트는 새로운 iSeries 소켓 기능적 확장 기능의 일부를 설명합니다.

IPv6 지원

이 릴리스에서 새롭게 지원되는 것으로 프로그래머는 이제 새로운 AF_INET6 주소 패밀리를 사용하여 IPv6 주소를 사용하는 어플리케이션을 작성할 수 있습니다. AF_INET6 주소 패밀리 외에 기존의 API도 새로운 주소 패밀리를 지원하도록 변경되었으며 새로운 API가 추가되고 새로운 구조가 정의되었습니다.

- AF_INET6 주소 패밀리
  이 주제에서는 새로운 주소 패밀리 및 주소 구조를 설명합니다.
- 시나리오: IPv4 및 IPv6 클라이언트를 허용하기 위한 어플리케이션 작성
  이 주제에서는 프로그래머가 IPv6 연결을 허용하기 위해 AF_INET6 주소 패밀리를 사용하는 어플리케이션을 설계하고 작성하려고 하는 고객 상황을 설명합니다. 이 주제에는 프로그래머가 자신의 요구에 맞게 변경할 수 있는 샘플 프로그램도 포함되어 있습니다.

X/Open Single UNIX® 스펙 호환성

OS/400 소켓은 X/Open Single UNIX 스펙을 사용하여 호환성을 지원합니다. OS/400 소켓은 이런 스펙과 일치하도록 구조, 유형 정의 및 기능 프로토타입을 변경하고 있습니다. 프로그래머는 BSD(Berkeley Software Distributions) 4.3 소켓 기반의 디폴트 OS/400 소켓을 사용하거나 _XOPEN_SOURCE 매크로를 지정하여 UNIX 98 호환 인터페이스를 선택할 수 있습니다.

- UNIX 98 호환성
  이 주제에서는 UNIX 98 소켓과 기본 OS/400 소켓간 차이점과 호환성을 설명합니다.

보안 소켓 층 성능 향상

이 릴리스에서는 보안 연결을 가속화하기 위해 새롭게 몇 가지 성능이 향상되었습니다. 현재 어플리케이션 프로그래머는 iSeries에서 다음 세 개의 별도 인터페이스를 통해 SSL 지원에 액세스할 수 있습니다.

1. 글로벌 보안 툴킷(GSKit) API
2. SSL_ APIs
3. Java SSL 인터페이스

Xsocket 툴 갱신

이 릴리스의 새로운 사항으로, QUSRTOOL과 함께 제공되는 대화식 툴인 Xsockets에 웹 브라우저 인터페이스를 통해 액세스할 수 있습니다. 새로운 지침이 추가되어 웹 브라우저를 사용하도록 툴을 설정하는 방법을 보

| 여줍니다. 여전히 명령행에서 툴에 대한 작업을 할 수는 있지만 이 새로운 인터페이스를 통해서만 새로운 기능
| (예: GSKit)에 액세스할 수 있습니다. 이런 변경사항에 대한 자세한 내용은 다음 주제를 참조하십시오.

| • Xsocket 툴

| IPX/SPX 및 AF_NS 주소 패밀리 제거

| V5R2에서 IBM은 IPX 및 SPX 프로토콜을 더 이상 지원하지 않습니다. 따라서 V5R2에서는 AF_NS 주소
| 패밀리가 작동 불가능합니다. 어플리케이션이 **socket()** API를 사용하여 AF_NS(IPX/SPX) 소켓을 열려고 시
| 도하면 -1 리턴 오류가 발생하고 errno가 EAFNOSUPPORT로 설정됩니다.

제 2 장 이 주제 인쇄

열람하거나 인쇄할 이 문서의 PDF 버전을 열람하거나 다운로드할 수 있습니다. PDF 파일을 보려면 Adobe® Acrobat® Reader를 설치해야 합니다. http://www.adobe.com/prodindex/acrobat/readstep.html  에서 사본을 다운로드할 수 있습니다.

PDF 버전을 보거나 다운로드하려면 소켓 프로그래밍을 선택하십시오(444KB 및 132 페이지).

보거나 인쇄하기 위해 PDF를 워크스테이션에 저장하려면 다음을 수행하십시오.

1. 브라우저에서 PDF를 여십시오(위의 링크를 클릭하십시오).
2. 브라우저 메뉴에서 **파일**을 클릭하십시오.
3. **다른 이름으로 저장...**을 클릭하십시오.
4. PDF를 저장할 디렉토리로 이동하십시오.
5. **저장**을 클릭하십시오.

제 3 장 소켓 프로그래밍의 전제조건

소켓 어플리케이션을 작성하기 전에 먼저 다음 단계를 완료해야 합니다.

컴파일러 요구사항

1. QSYSINC 라이브러리를 설치하십시오. 이 라이브러리는 소켓 어플리케이션 컴파일시 필요한 필수 헤더 파일을 제공합니다.
2. C 컴파일러 사용권 프로그램(5722-CX2)을 설치하십시오.

AF_INET 및 AF_INET6 주소 패밀리에 대한 요구사항

컴파일러 요구사항 이외에 다음을 완료해야 합니다.

1. TCP/IP 계획
2. TCP/IP 설치
3. 처음 TCP/IP 구성
4. IPv6에 TCP/IP 구성. 이 단계는 선택적입니다. AF_INET6 주소 패밀리를 사용하는 어플리케이션을 작성할 경우 TCP/IP에 IPv6 인터페이스를 구성하십시오.

보안 소켓 층(SSL) 및 글로벌 보안 툴킷(GSKit) API 요구사항

컴파일러와 AF_INET 및 AF_INET6 주소 요구사항 이외에 다음 타스크를 완료하여 보안 소켓에 대한 작업을 해야 합니다.

1. 디지털 인증 관리자 사용권 프로그램(5722-SS1 옵션 34)을 설치하고 구성하십시오. 자세한 내용은 Information Center의 디지털 인증 관리자를 참조하십시오.
2. 암호 액세스 제공자 사용권 프로그램(5722- AC3).
3. SSL과 암호 하드웨어를 함께 사용하려는 경우 iSeries용 2058 Cryptographic Accelerator 또는 iSeries용 4758 PCI Cryptographic Coprocessor를 설치하고 구성해야 합니다. 2058 Cryptographic Accelerator를 사용하면 오퍼레이팅 시스템에서 카드로 SSL 암호 처리를 오프로드할 수 있습니다. 2058 Cryptographic Accelerator 및 그 피처에 대한 자세한 설명은 iSeries용 2058 Cryptographic Accelerator를 참조하십시오. 4758 Cryptographic Coprocessor는 SSL 암호 처리에 사용할 수 있습니다. 그러나 2058과 달리 이 카드는 암호화 및 암호해독 키 같은 보다 많은 암호 기능을 제공합니다. 이 카드의 피처 및 구성 단계는 iSeries용 4758 PCI Cryptographic Coprocessor를 참조하십시오.

AF_TELEPHONY 주소 패밀리에 대한 요구사항

전화 회선을 사용하는 AF_TELEPHONY 소켓을 설계하려면 일반적인 요구사항 외에도 다음 단계를 완료해야 합니다.

1. IDSN 서비스를 계획하여 ISDN 연결을 해야 하는 필요성을 판별하십시오.
2. 계획 정보를 기반으로 IDSN 환경을 구성하십시오.

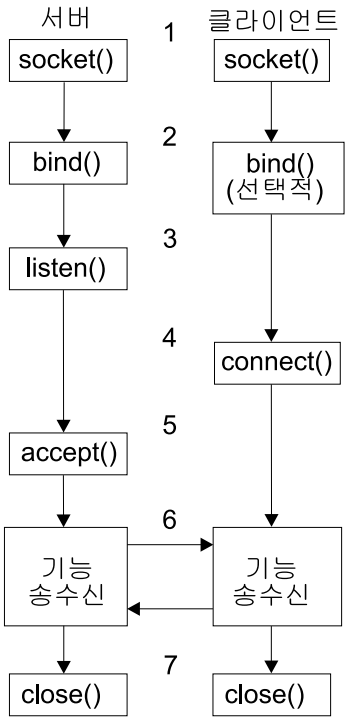
제 4 장 소켓 작동 방법

소켓은 일반적으로 클라이언트/서버 상호작용에 사용됩니다. 일반적인 시스템 구성에서는 서버를 하나의 기계에 배치하고 클라이언트가 다른 기계에 배치합니다. 클라이언트는 서버로 연결하여 정보를 교환한 후 연결을 해제합니다.

소켓에는 일반적인 이벤트 흐름이 있습니다. 연결 지향의 클라이언트 대 서버 모델에서 서버 프로세스의 소켓은 클라이언트의 요청을 기다립니다. 이를 위해 서버는 우선 클라이언트가 서버를 찾는 데 사용할 주소를 설정(바인드)합니다. 일단 주소가 설정되면 서버는 클라이언트가 서비스를 요청할 때까지 기다립니다. 클라이언트가 소켓을 통해 서버로 연결되면 클라이언트 대 서버 자료 교환이 이루어집니다. 서버는 클라이언트 요구를 수행하고 응답을 다시 클라이언트로 송신합니다.

주: 현재 IBM은 대부분의 소켓 API의 두 가지 버전을 지원합니다. 디폴트 OS/400 소켓은 BSD(Berkeley Socket Distribution) 4.3 구조 및 구문을 사용합니다. 기본 OS/400 소켓과 BSD 4.3간의 차이점은 BSD(Berkeley Socket Distribution) 호환성에 간략히 설명되어 있습니다. 다른 소켓 버전은 BSD 4.4 및 UNIX 98 프로그래밍 인터페이스 스펙과 호환 가능한 구문 및 구조를 사용합니다. 프로그래머는 _XOPEN_SOURCE 매크로를 지정하여 UNIX98 호환 인터페이스를 사용할 수 있습니다. 이런 API 및 구조 차이점에 대한 설명은 UNIX 98 호환성을 참조하십시오.

다음 그림은 연결 지향 소켓 세션에 대한 일반적인 이벤트 흐름(및 발행된 함수 순서)을 보여줍니다. 그림 다음에는 각 이벤트에 대한 설명이 제시됩니다.



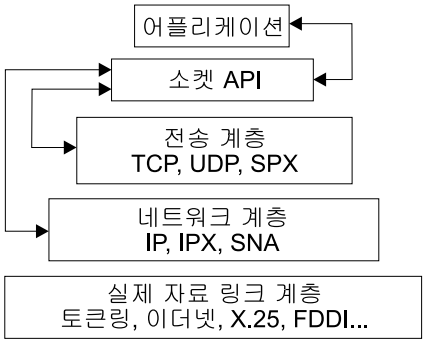
연결 지향 소켓에 대한 이벤트의 일반적인 흐름

1. **socket()** 함수는 통신 종료점을 작성한 후 해당 종료점을 나타내는 소켓 설명자를 리턴합니다.

2. 소켓 설명자가 있는 어플리케이션은 고유한 이름을 소켓에 바인드할 수 있습니다. 서버는 네트워크에서 액세스 가능한 이름을 바인드해야 합니다.
3. **listen()** 함수는 클라이언트 연결 요청을 승인할 것임을 표시합니다. 소켓에 대한 **listen()**이 발행되면 해당 소켓은 연결 요청을 능동적으로 개시할 수 없습니다. **socket()** 함수로 소켓이 할당되고 **bind()** 함수로 소켓에 이름이 바인드된 후 **listen()** 함수가 발행됩니다. **listen()** 함수는 **accept()** 함수가 발행되기 전에 발행되어야 합니다.
4. 클라이언트 어플리케이션은 스트림 소켓에서의 **connect()** 함수를 사용하여 서버에 대한 연결을 설정합니다.
5. 서버 어플리케이션에서는 클라이언트 연결 요구를 승인하기 위해 **accept()** 함수를 사용합니다. 서버는 **accept()**를 발행하기 전에 **bind()** 및 **listen()** 함수를 발행해야 합니다.
6. 스트림 소켓간(클라이언트와 서버간) 연결이 설정되면 모든 소켓 API 자료 전송 함수를 사용할 수 있습니다. 클라이언트와 서버에는 **send()**, **recv()**, **read()**, **write()** 등과 같은 여러 가지 다양한 자료 전송 함수가 있습니다.
7. 서버나 클라이언트가 연산을 종료하려면 **close()** 함수를 발행하여 소켓이 취한 모든 시스템 자원을 해제해야 합니다.

주:

소켓 API는 어플리케이션층과 전송층 사이의 통신 모델에 위치해 있습니다. 소켓 API는 통신 모델에 있는 계층이 아닙니다. 소켓 API에서는 어플리케이션이 일반 통신 모델의 전송층이나 네트워크층과 상호작용할 수 있습니다. 다음 그림의 화살표는 소켓의 위치와 소켓이 제공하는 통신층을 보여줍니다.



일반적으로, 네트워크 구성은 보안 내부 네트워크와 비보안 외부 네트워크 사이의 연결을 허용하지 않습니다. 그러나 소켓이 방화벽 외부의 시스템(보안이 매우 우수한 호스트)에서 실행되는 서버 프로그램과 통신하도록 할 수 있습니다.

또한, 소켓은 MPTN(Multiprotocol Transport Networking) 구조에 대한 IBM의 AnyNet 구현의 일부가 됩니다. MPTN 구조는 추가 전송 네트워크를 통한 전송 네트워크를 조작하고 서로 다른 유형의 전송 네트워크를 통해 어플리케이션 프로그램을 연결할 수 있는 기능을 제공합니다.

Information Center를 사용하여 몇 가지 방법으로 API 참조 정보에 액세스할 수 있습니다. 소켓 API 주제는 소켓 기능 및 구조에 대한 개요를 제공합니다. 특정 API를 탐색하거나 API 범주를 탐색하려는 경우 API 정

| 보는 대화식 API 파인더를 제공하여 사용자를 지원합니다.

제 5 장 소켓 특성

소켓은 다음 특성을 공유합니다.

- 소켓은 정수로 표시됩니다. 그러한 정수를 **소켓 설명자**라고 합니다.
- 소켓은 프로세스가 소켓으로의 열린 링크를 유지하는 한 존재합니다.
- 소켓 이름을 지정한 후 통신 정의역 내의 다른 소켓과의 통신에 사용할 수 있습니다.
- 소켓은 서버가 소켓으로부터의 연결을 승인하거나 소켓과 메세지를 교환할 때 통신을 수행합니다.
- 쌍으로 소켓을 작성할 수 있습니다(AF_UNIX 주소 패밀리의 소켓에 대해서만).

소켓이 제공하는 연결은 연결 지향이거나 무접속일 수 있습니다. **연결 지향** 통신이란 연결이 설정되고 프로그램 사이의 대화가 뒤따르는 것을 의미합니다. 서비스를 제공하는 프로그램(서버 프로그램)은 수신 연결 요구를 승인할 수 있는 사용 가능한 소켓을 설정합니다. 선택적으로, 서버는 클라이언트에서 해당 서비스를 얻는 위치와 해당 서비스에 연결하는 위치를 식별할 수 있는 이름을 제공되는 서비스에 지정할 수 있습니다. 서비스 클라이언트(클라이언트 프로그램)는 서버 프로그램에게 서비스를 요청해야 합니다. 클라이언트는 고유한 이름이나 서버 프로그램이 지정한 고유한 이름과 연관된 속성에 연결하여 이를 수행합니다. 이는 전화번호(ID)를 다이얼한 후 서비스(예: 배관공)를 제공하는 다른 상대방과 연결하는 것과 같은 방식입니다. 호출 리시버(서버, 이 예에서는 배관공)가 전화에 응답하면 연결이 설정됩니다. 올바른 상대방에 연결되었는지 그리고 호출자와 상대방이 연결을 원하는 동안 연결이 활동 상태로 유지되는지를 확인합니다.

무접속 통신이란 대화나 자료 전송이 발생하는 동안 연결이 설정되지 않는 것을 말합니다. 그 대신, 서버 프로그램은 도달 장소(예: 우체통)를 식별하는 이름을 지정합니다. 우체통으로 편지를 보내면 리시버가 편지를 받았는지 확인할 수 없습니다. 편지에 대한 응답이 필요한 경우도 있습니다. 자료가 교환되는 활동 중인 실시간 연결이 없습니다.

소켓 특성 판별 방법

어플리케이션이 **socket()** 함수로 소켓을 작성할 때 다음과 같은 매개변수를 지정하여 소켓을 식별해야 합니다.

- 소켓 주소 패밀리는 소켓의 주소 구조 형식을 판별합니다. 이 주제에는 각 주소 패밀리의 주소 구조의 예가 나와 있습니다. 소켓 주소 구조의 일반적인 정의는 소켓 주소 구조를 참조하십시오.
- 소켓 유형은 소켓의 필요한 통신 양식을 판별합니다.
- 소켓 지원 프로토콜은 소켓에서 사용하는 프로토콜을 판별합니다.

이러한 매개변수나 특성은 소켓 어플리케이션과 기타 소켓 어플리케이션과 상호작용하는 방법을 정의합니다. 소켓의 주소 패밀리에 따라, 소켓 유형 및 프로토콜을 다르게 선택할 수 있습니다. 다음 테이블은 해당 주소 패밀리와 그 연관 소켓 유형 및 프로토콜을 보여줍니다.

표 1. 소켓 특성 요약

주소 패밀리	소켓 유형	소켓 프로토콜
AF_UNIX	SOCK_STREAM	N/A
	SOCK_DGRAM	N/A
AF_INET	SOCK_STREAM	TCP
	SOCK_DGRAM	UDP
	SOCK_RAW	IP, ICMP
AF_INET6	SOCK_STREAM	TCP
	SOCK_DGRAM	UDP
	SOCK_RAW	IP6, ICMP6
AF_TELEPHONY	SOCK_STREAM	N/A
AF_UNIX_CCSID	SOCK_STREAM	N/A
	SOCK_DGRAM	N/A

이러한 소켓 특성이나 매개변수외에도, 상수값은 QSYSINC 라이브러리와 함께 제공된 네트워크 루틴 및 헤더 파일에 정의됩니다. 헤더 파일의 설명의 경우 Information Center의 소켓 API 주제에 나열된 개별 API를 참조하십시오. 각 API는 API 설명의 사용 섹션에 해당 헤더 파일을 나열합니다.

소켓 네트워크 루틴을 사용하여 DNS, 호스트, 프로토콜, 서비스 및 네트워크 파일에서 정보를 얻을 수 있습니다. 이러한 루틴에 대한 설명은 소켓 네트워크 루틴을 참조하십시오.

소켓 주소 구조

소켓은 **sockaddr** 주소 구조를 사용하여 주소를 전달하고 수신합니다. 이 구조에서는 소켓 API가 주소지정 형식을 인식하지 않아도 됩니다. 현재 OS/400은 BSD(Berkeley Software Distributions) 4.3 및 X/Open Single Unix 스펙(UNIX 98)을 지원합니다. 기본 OS/400 API는 BSD 4.3 구조 및 구문을 사용합니다. _XOPEN_SOURCE 매크로 값을 520 이상으로 정의하면 UNIX 98 호환 인터페이스를 선택할 수 있습니다. 사용되는 BSD 4.3에 대해 각 소켓 주소는 동등한 UNIX 98 구조를 갖습니다.

표 2. BSD 4.3 및UNIX 98/BSD 4.4 소켓 주소 구조 비교

BSD 4.3 구조	**BSD 4.4/UNIX 98** 호환 구조

`struct sockaddr{`
`  u_short sa_family;`
`  char  sa_data [14];`
`};`
 	
`struct sockaddr_storage{`
`  sa_family_t  ss_family;`
`  char  _ss_pad1[_SS_PAD1SIZE];`
`  char*  _ss_align;`
`  char  _ss_pad2[_SS_PAD2SIZE];`
`};`
	`  struct sockaddr {`
	` uint8_t  sa_len;`
	` sa_family_t sa_family`
	`  char  sa_data[14]`
	`};`
	
	`struct sockaddr_storage {`
	`  uint8_t  ss_len;`
	`  sa_family_t  ss_family;`
	`  char  _ss_pad1[_SS_PAD1SIZE];`
	`  char*  _ss_align;`
	`  char  _ss_pad2[_SS_PAD2SIZE];`
	`};`

표 3. 주소 구조화

주소 구조화 필드	정의
sa_len	이 필드는 UNIX 98 스펙의 주소 길이를 포함합니다. **주:** sa_len 필드는 BSD 4.4 호환성을 위해 유일하게 제공됩니다. BSD 4.4/UNIX 98 호환성을 사용할 경우에는 이 필드를 사용하지 않아도 됩니다. 이 필드는 입력 주소에서는 무시됩니다.
sa_family	이 필드에는 주소 패밀리가 정의됩니다. 이 값은 **socket()** 호출시 주소 패밀리에 지정됩니다.
sa_data	이 필드에는 주소 자체를 보유하기 위해 14바이트가 예약되어 있습니다. **주:** 14바이트의 sa data 길이는 주소에 대한 위치 홀더입니다. 실제 주소는 이 길이를 초과해도 됩니다. 이 구조는 주소 형식을 정의하지 않는다는 점에서도 총칭적입니다. 주소 형식은 소켓이 작성된 전송 유형으로 정의됩니다. 각 전송 제공자는 유사한 주소 구조에 특정 주소지정 요구사항의 정확한 형식을 정의합니다. 전송은 **socket()** API에서 프로토콜 매개변수 값으로 식별됩니다.
sockaddr_storage	주소 패밀리에 대한 기억장치를 선언합니다. 이 구조는 상당히 크며 프로토콜 고유 구조에 대해 배열됩니다. 그런 다음 sockaddr 구조로 캐스트되어 API에서 사용할 수 있습니다. sockaddr_storage의 ss_family 필드는 항상 프로토콜 고유 구조의 패밀리 필드와 함께 배열됩니다.

소켓 주소 패밀리

socket()의 주소 패밀리 매개변수는 소켓 함수에서 사용할 주소 구조의 형식을 판별합니다. 주소 패밀리 프로토콜은 한 어플리케이션에서 다른 어플리케이션으로(또는 동일한 기계 내의 한 프로세스에서 다른 프로세스로) 어플리케이션 자료의 네트워크 전송을 제공합니다. 어플리케이션은 소켓의 프로토콜 매개변수에서 네트워크 전송 제공자를 지정합니다.

socket() 함수의 주소 패밀리 매개변수(address_family)는 소켓 함수에 사용되는 주소 구조를 지정합니다. 다음 주제에는 이런 주소 패밀리 각각, 주소 패밀리의 사용, 관련 프로토콜 및 관련 구조의 예가 설명되어 있습니다.

- AF_INET 주소 패밀리
- AF_INET6 주소 패밀리
- AF_UNIX 주소 패밀리
- AF_UNIX_CCSID 주소 패밀리
- AF_TELEPHONY 주소 패밀리

AF_INET 주소 패밀리

이 주소 패밀리에서는 동일한 시스템이나 다른 시스템에서 실행되는 프로세스 사이의 상호 프로세스 통신을 제공합니다. AF_INET 소켓용 주소는 IP 주소와 포트 번호입니다. 130.99.128.1과 같은 IP 주소로 또는 해당 32비트 양식 X‘82638001’에서 AF_INET 소켓의 IP 주소를 지정할 수 있습니다.

인터넷 프로토콜 버전 4(IPv4)를 사용하는 소켓 어플리케이션의 경우 AF_INET 주소 패밀리는 **sockaddr_in** 주소 구조를 사용합니다. _XOPEN_SOURCE 매크로를 사용할 때 AF_INET 주소 구조는 BSD 4.4/UNIX 98 스펙과 호환되도록 변경됩니다. sockaddr_in 주소 구조의 경우 이런 차이점이 다음 표에 요약되어 있습니다.

표 4. sockaddr_in 주소 구조의 경우 BSD 4.3과 BSD 4.4/UNIX 98간 차이점

BSD 4.3 sockaddr_in 주소 구조화	**BSD 4.4/UNIX 98 sockaddr_in 주소 구조화**

`struct sockaddr_in {`
`  short  sin_family;`
`  u_short sin_port;`
`  struct  in_addr sin_addr;`
`  char sin_zero[8];`
`};`
	`struct sockaddr_in {`
	`  uint8_t  sin_len;`
	`  sa_family_t  sin_family;`
	`  u_short sin_port;`
	`  struct  in_addr sin_addr;`
	`  char sin_zero[8];`
	`};`

표 5. AF_INET 주소 구조

주소 구조화 필드	정의
sin_len	이 필드는 UNIX 98 스펙의 주소 길이를 포함합니다. **주:** sa_len 필드는 BSD 4.4 호환성을 위해 유일하게 제공됩니다. BSD 4.4/UNIX 98 호환성을 사용할 경우에는 이 필드를 사용하지 않아도 됩니다. 이 필드는 입력 주소에서는 무시됩니다.
sin_family	이 필드에는 TCP 또는 UDP 사용시 항상 AF_INET인 주소 패밀리가 들어 있습니다.
sin_port	이 필드에는 포트 번호가 들어 있습니다.
sin_addr	이 필드에는 인터넷 주소가 들어 있습니다.
sin_zero	이 필드는 예약되어 있습니다. 이 필드를 16진 0으로 설정하십시오.

AF_INET 및 AF_INET 주소 패밀리를 사용하는 샘플 프로그램 사용시 정보를 보려면 AF_INET 주소 패밀리 사용을 참조하십시오.

AF_INET6 주소 패밀리

이 주소 패밀리는 인터넷 프로토콜 버전 6(IPv6)을 지원합니다. AF_INET6 주소 패밀리는 128비트(16바이트) 주소를 사용합니다. 이런 주소의 기본 구조에는 네트워크 번호에 64비트, 호스트 번호에 다시 64비트가 포함됩니다. AF_INET6 주소를 x:x:x:x:x:x:x:x로 지정할 수 있는데 여기서 'x'는 8개의 16비트 주소 단편에 대한 16진 값입니다. 예를 들어, FEDC:BA98:7654:3210:FEDC:BA98:7654:3210은 유효한 주소입니다.

TCP, UDP 또는 RAW를 사용하는 소켓 어플리케이션의 경우 AF_INET6 주소 패밀리는 **sockaddr_in6** 주소 구조를 사용합니다. _XOPEN_SOURCE 매크로를 사용하여 BSD 4.4/UNIX 98 스펙을 구현하는 경우 이 주소 구조가 변경됩니다. sockaddr_in6 주소 구조의 경우 이런 차이점이 다음 표에 요약되어 있습니다.

표 6. sockaddr_in6 주소 구조의 경우 BSD 4.3과 BSD 4.4/UNIX 98간 차이점

BSD 4.3 sockaddr_in6 주소 구조	**BSD 4.4/UNIX 98 sockaddr_in6 주소 구조**

표 6. *sockaddr_in6 주소 구조의 경우 BSD 4.3과 BSD 4.4/UNIX 98간 차이점 (계속)*

struct sockaddr_in6 {
sa_family_t sin6_family;
in_port_t sin6_port;
uint32_t sin6_flowinfo;
struct in6_addr sin6_addr;
uint32_t sin6_scope_id;
};
	struct sockaddr_in6 {
	uint8_t sin6_len;
	sa_family_t sin6_family;
	in_port_t sin6_port;
	uint32_t sin6_flowinfo;
	struct in6_addr sin6_addr;
	uint32_t sin6_scope_id;
	};

표 7. *AF_INET6 주소 구조*

주소 구조화 필드	정의
sin6_len	이 필드는 UNIX 98 스펙의 주소 길이를 포함합니다. **주:** sin6_len 필드는 BSD 4.4 호환성을 위해 유일하게 제공됩니다. BSD 4.4/UNIX 98 호환성을 사용할 경우에는 이 필드를 사용하지 않아도 됩니다. 이 필드는 입력 주소에서는 무시됩니다.
sin6_family	이 필드는 AF_INET6 주소 패밀리를 지정합니다.
sin6_port	이 필드에는 전송층 포트가 들어 있습니다.
sin6_flowinfo	이 필드에는 두 개의 정보 단편(통신 클래스와 흐름 레이블)이 포함됩니다. **주:** 현재 이 필드는 지원되지 않으며 상위 호환성을 위해 0으로 설정해야 합니다.
sin6_addr	이 필드에는 IPv6 주소가 지정됩니다.
sin6_scope_id	이 필드는 일련의 인터페이스를 **sin6_addr** 필드에 기재된 주소 범위에 적절한 것으로 식별합니다. **주:** 현재 이 필드는 지원되지 않으며 상위 호환성을 위해 0으로 설정해야 합니다.

AF_UNIX 주소 패밀리

이 주소 패밀리는 소켓 API를 사용하는 동일한 시스템에서 상호 프로세스 통신을 제공합니다. 주소는 실제로 파일 시스템의 항목에 대한 경로명입니다. 루트 디렉토리나 모든 열린 파일 시스템에서 그러나 QSYS 또는 QDOC와 같은 파일 시스템에서 루트 디렉토리에 소켓을 작성할 수 있습니다. 데이터그램을 다시 수신하려면 프로그램에서는 AF_UNIX, SOCK_DGRAM 소켓을 이름에 바인드해야 합니다. 또한 프로그램에서는 소켓이 닫힐 때 **unlink()** API로 파일 시스템 오브젝트를 명시적으로 제거해야 합니다.

주소 패밀리가 AF_UNIX인 소켓은 **sockaddr_un** 주소 구조를 사용합니다. _XOPEN_SOURCE 매크로를 사용하여 BSD 4.4/UNIX 98 스펙을 구현하는 경우 이 주소 구조가 변경됩니다. sockaddr_un 주소 구조의 경우 이런 차이점은 다음 표에 요약되어 있습니다.

표 8. *sockaddr_un 주소 구조의 경우 BSD 4.3과 BSD 4.4/UNIX 98간 차이점*

BSD 4.3 sockaddr_un 주소 구조	**BSD 4.4/UNIX 98 sockaddr_un 주소 구조**

struct sockaddr_un {
short sun_family;
char sun_path[126];
};
	struct sockaddr_un {
	uint8_t sun_len;
	sa_family_t sun_family;
	char sun_path[126];
	};

| *표 9. AF_UNIX 주소 구조*

주소 구조화 필드	정의
sun_len	이 필드는 UNIX 98 스펙의 주소 길이를 포함합니다. **주:** sun_len 필드는 BSD 4.4 호환성을 위해 유일하게 제공됩니다. BSD 4.4/UNIX 98 호환성을 사용할 경우에는 이 필드를 사용하지 않아도 됩니다. 이 필드는 입력 주소에서는 무시됩니다.
sun_family	이 필드에는 주소 패밀리가 들어 있습니다.
sun_path	이 필드에는 파일 시스템의 항목에 대한 경로 이름이 들어 있습니다.

AF_UNIX 주소 패밀리의 경우 프로토콜 표준이 포함되지 않으므로 프로토콜 스펙이 적용되지 않습니다. 두 프로세스에서 사용하는 통신 메카니즘은 시스템에 따라 다릅니다.

AF_UNIX 사용에 대한 정보는 AF_UNIX 주소 패밀리 사용을 참조하십시오. 여기에는 이 주소 패밀리를 사용하는 샘플 프로그램이 나와 있습니다.

AF_UNIX_CCSID 주소 패밀리

AF_UNIX_CCSID 패밀리는 AF_UNIX 주소 패밀리와 호환성이 있으며 동일한 제한사항이 있습니다. 둘 모두 무접속 또는 연결 지향일 수 있으므로 두 프로세스를 연결하는 외부 통신 기능이 없습니다. 차이점은 주소 패밀리가 AF_UNIX_CCSID인 소켓이 **sockaddr_unc** 주소 구조를 사용한다는 점입니다. 이 주소 구조는 sockaddr_un과 유사하나 **Qlg_Path_Name_T** 형식을 사용하여 UNICODE의 경로명이나 CCSID를 허용합니다. Information Center의 경로명 형식을 참조하십시오.

그러나 AF_UNIX 소켓이 AF_UNIX 주소 구조의 AF_UNIX_CCSID 소켓에서 경로명을 리턴할 수 있으므로, 경로 크기가 제한됩니다. AF_UNIX는 126자만 지원하므로 AF_UNIX_CCSID도 126자로 제한됩니다.

사용자는 하나의 소켓에서 AF_UNIX 및 AF_UNIX_CCSID 주소를 교환할 수 없습니다. AF_UNIX_CCSID가 **socket()** 호출에 지정되면 모든 주소가 차후 API 호출시 **sockaddr_unc**이어야 합니다.

```
struct sockaddr_unc {
     short             sunc_family;
     short             sunc_format;
     char              sunc_zero[12];
     Qlg_Path_Name_T   sunc_qlg;
     union {
     char              unix[126];
     wchar_t           wide[126];
     char*             p_unix;
     wchar_t*     p_wide;
    }                  sunc_path;
    } ;
```

표 10. AF_UNIX_CCSID 주소 구조

주소 구조화 필드	정의
sunc_family	이 필드에는 항상 AF_UNIX_CCSID인 주소 패밀리가 들어 있습니다.

표 10. AF_UNIX_CCSID 주소 구조 (계속)

sunc_format	이 필드에는 경로명의 형식에 정의된 두 개의 값이 들어 있습니다. • **SO_UNC_DEFAULT**는 통합 파일 시스템 경로명의 현재 디폴트 CCSID를 사용하는 광범위한 경로명을 나타냅니다. sunc_qlg 필드는 무시됩니다. • **SO_UNC_USE_QLG**는 **sunc_qlg** 필드가 경로명의 형식 및 CCSID를 정의한다는 것을 나타냅니다.
sunc_zero	이 필드는 예약되어 있습니다. 이 필드를 16진 0으로 설정하십시오.
sunc_qlg	이 필드는 경로명 형식을 지정합니다.
sunc_path	이 필드에는 경로명이 들어 있습니다. 최대 126자이며 1바이트 또는 2바이트일 수 있습니다. sunc_path 필드 내에 포함되거나 별도로 할당된 후 sunc_path에 의해 지정될 수 있습니다. 형식은 sunc_format 및 sunc_qlg로 판별됩니다.

AF_UNIX_CCSID 소켓에 대한 자세한 정보는 샘플 프로그램을 제공하는 AF_UNIX_CCSID 주소 패밀리 사용을 참조하십시오.

AF_TELEPHONY 주소 패밀리

AF_TELEPHONY 주소 패밀리를 사용하여 사용자는 표준 소켓 API를 사용하는 ISDN 전화 네트워크를 통해 전화를 걸고 받을 수 있습니다. 이 정의역에서 연결의 종료점을 형성하는 소켓은 실제로 전화 호출의 피호출 및 호출 상대입니다. 이 주소 패밀리의 주소는 40자리 전화 번호로 표시됩니다. 이 주소 패밀리는 팩스 지원 개발시 가장 일반적으로 사용됩니다.

시스템에서는 소켓 유형이 SOCK_STREAM인 연결 지향 소켓으로 AF_TELEPHONY 소켓만 지원합니다. 전화 통신 정의역 소켓의 연결은 신뢰성이 기초 전화 연결보다 더 높지는 않습니다. 전달을 보증하려면 이 주소 패밀리를 사용하는 팩스 어플리케이션과 같은 이러한 서비스를 제공하는 어플리케이션에 대한 작업을 해야 합니다.

주소 패밀리가 AF_TELEPHONY인 소켓은 **sockaddr_tel** 주소 구조를 사용합니다.

```
      struct sockaddr_tel {
         short             stel_family;
         struct tel_addr   stel_addr;
     char stel_zero[4];
};
```

전화 주소는 뒤에 최대 40자리의 전화번호(0 - 9)가 붙는 2바이트 길이로 구성됩니다.

```
      struct tel_addr {
     unsigned short t_len
     char t_addr[40];
};
```

표 11. AF_TELEPHONY 주소 구조

주소 구조화 필드	정의
stel_family	이 필드에는 주소 패밀리가 들어 있습니다.

표 11. AF_TELEPHONY 주소 구조 (계속)

stel_addr	이 필드에는 전화 통신 주소가 들어 있습니다.
stel_zero	이 필드는 예약된 필드입니다.

AF_TELEPHONY 주소 패밀리에 대한 정보는 AF_TELEPHONY 주소 패밀리 사용을 위한 환경 구성의 단계를 제공하는 AF_TELEPHONY 주소 패밀리 사용을 참조하십시오.

소켓 유형

소켓 호출시 두 번째 매개변수는 소켓의 유형을 결정합니다. 소켓 유형은 한 기계에서 또 다른 기계로 또는 한 프로세스에서 또 다른 프로세스로 자료 전송에 가능한 연결의 유형 및 특성을 식별합니다. 다음 리스트는 iSeries에서 지원하는 소켓 유형을 설명합니다.

스트림(SOCK_STREAM)
이 소켓 유형은 연결 지향입니다. 즉, **bind()**, **listen()**, **accept()**, **connect()** 함수를 사용하여 종단간 연결 설정하십시오. SOCK_STREAM은 오류 또는 중복 없이 자료를 송신하고 송신 순서대로 그 자료를 수신합니다. SOCK_STREAM은 자료 과수행을 피하기 위해 흐름 제어를 빌드합니다. 자료의 레코드 경계는 부과하지 않습니다. SOCK_STREAM은 자료를 바이트 스트림으로 간주합니다. iSeries 구현에서 전송 제어 프로토콜(TCP), 시스템 네트워크 구조(SNA), AF_UNIX, AF_UNIX_CCSID 및 AF TELEPHONY 소켓을 통해 스트림 소켓을 사용할 수 있습니다. 스트림 소켓을 사용하여 보안 호스트(방화벽) 외부의 시스템과 통신할 수도 있습니다.

데이터그램(SOCK_DGRAM)
인터넷 프로토콜 용어로 자료 전송의 기본 단위를 **데이터그램**이라고 합니다. 이것은 기본적으로 뒤에 일부 자료가 오는 헤더입니다. 데이터그램 소켓은 무접속입니다. 전송 제공자(프로토콜)에 대해 종단간 연결을 설정하지 않습니다. 소켓은 데이터그램을 전달 보증없이 독립적인 패킷으로 송신합니다. 자료가 유실되거나 중복될 수 있습니다. 데이터그램은 순서대로 도달하지 않을 수 있습니다. 데이터그램의 *크기*는 단일 트랜잭션으로 송신될 수 있는 자료 *크기*로 제한됩니다. 일부 전송 제공자의 경우 각 데이터그램은 네트워크를 통해 서로 다른 라우트 경로를 사용할 수 있습니다. 이러한 유형의 소켓에서 **connect()** 함수를 발행할 수 있습니다. 그러나 **connect()** 함수에서 프로그램이 송신 및 수신하는 목적지 주소를 지정해야 합니다. iSeries 구현에서 사용자 데이터그램 프로토콜(UDP), SNA를 통해 데이터그램 소켓을 사용할 수 있으며 AF_UNIX 및 AF_UNIX_CCSID 주소 패밀리와 데이터그램 소켓을 함께 사용할 수 있습니다.

원시(SOCK_RAW)
이 유형의 소켓은 인터넷 프로토콜(IPv4 또는 IPv6) 및 인터넷 제어 메세지 프로토콜(ICMP 또는 ICMP6)과 같은 하위층 프로토콜에 대한 직접 액세스를 허용합니다. SOCK_RAW를 사용하려면 전송 제공자에 의해 사용되는 프로토콜 헤더 정보를 관리해야 하므로 프로그래밍 전문 지식이 더 요구됩니다. 이 레벨에서 전송 제공자는 각자 서로 다른 자료 형식 및 의미를 지정합니다.

소켓 프로토콜

프로토콜은 한 기계에서 다른 기계로(또는 동일한 기계 내의 한 프로세스에서 다른 프로세스로) 어플리케이션 자료의 네트워크 전송을 제공합니다. 어플리케이션은 **socket()** 함수의 **프로토콜** 매개변수에서 전송 제공자를 지정합니다.

AF_INET 주소 패밀리의 경우 둘 이상의 전송 제공자가 허용됩니다. SNA, TCP/IP 또는 UDP/IP 프로토콜은 동시에 동일한 소켓에서 활동할 수 있습니다. ALWANYNET(ANYNET 지원 허용) 네트워크 속성을 사용하여 고객은 AF_INET 소켓 어플리케이션에 TCP/IP가 아닌 전송을 사용할 수 있습니다. 이 네트워크 속성은 ***YES** 또는 ***NO**일 수 있습니다. 디폴트 값은 ***NO**입니다.

예를 들어, 현재 상태(디폴트 상태)가 *NO이면 SNA를 통한 AF_INET 전송은 사용되지 않습니다. AF_INET 소켓이 TCP/IP 전송만 통해 사용될 경우 ALWANYNET 상태는 CPU 활용을 향상시키기 위해 *NO로 설정되어야 합니다.

주: ALWANYNET 네트워크 속성은 TCP/IP를 통한 APPC에도 영향을 줍니다.

APPC 구성 옵션에 대한 정보는 APPC, APPN 및 HPR 구성을 참조하십시오.

TCP/IP를 통한 AF_INET 소켓은 소켓이 인터넷 프로토콜(IP)로 알려져 있는 네트워크층과 직접 통신한다는 것을 의미하는 SOCK_RAW의 유형을 지정할 수도 있습니다. TCP 또는 UDP 전송 제공자는 정상적으로 이 계층과 통신합니다. SOCK_RAW 소켓을 사용할 때 어플리케이션 프로그램은 0에서 255 사이의 프로토콜(TCP 및 UDP 프로토콜 제외)을 지정합니다. 이 프로토콜 번호는 기계가 네트워크상에서 통신할 때 IP 헤더 내에서 이동하게 됩니다. 결과적으로, 어플리케이션 프로그램은 UDP 또는 TCP 전송이 일반적으로 제공하는 모든 전송 서비스에 제공해야 하므로, 전송 제공자입니다.

AF_UNIX, AF_UNIX_CCSID 및 AF_TELEPHONY 주소 패밀리의 경우 프로토콜 스펙은 포함된 프로토콜 표준이 없으므로 실제로 중요하지 않습니다. 동일한 기계에서 두 프로세스간의 통신 메카니즘은 기계에 따라 다릅니다.

제 6 장 기본 소켓 설계

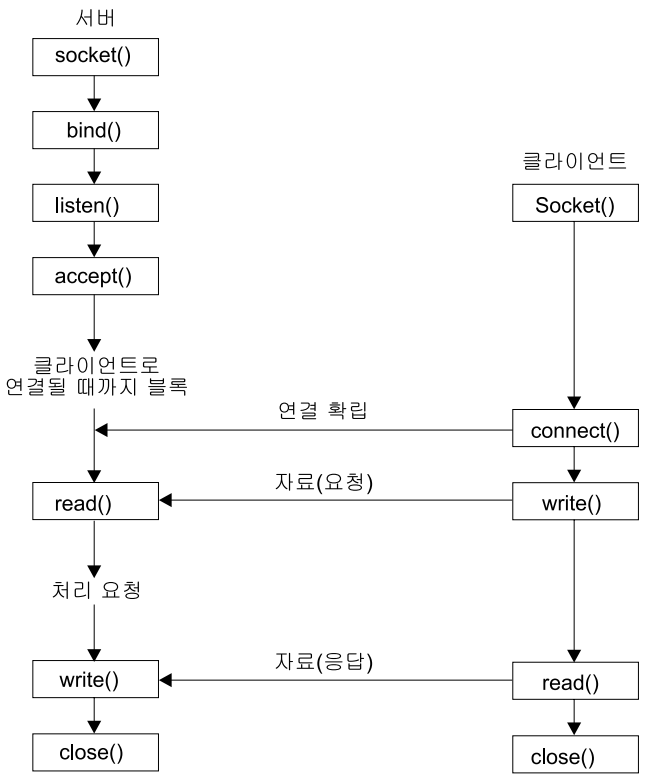
이 주제에는 가장 기본적인 설계를 사용하는 소켓 프로그램의 예가 나와 있습니다. 이러한 예는 보다 복잡한 소켓 설계의 기초를 제공합니다. 이들은 이전의 주제에서 강조표시된 일부 기본 개념을 구현합니다. 다음 샘플 프로그램은 소켓 프로그램의 가장 일반적인 유형의 예를 제공합니다.

- 연결 지향 소켓 작성
- 연결 소켓 작성
- 주소 패밀리를 사용하여 어플리케이션 설계

연결 지향 소켓 작성

다음 서버 및 클라이언트 예는 전송 제어 프로토콜(TCP)과 같은 연결 지향 프로토콜에 대해 작성된 소켓 API를 보여줍니다.

다음 그림은 연결 지향 프로토콜에 대한 소켓 API의 클라이언트/서버 관계를 보여줍니다.



이벤트의 소켓 흐름: 연결 지향 서버

다음 소켓 호출 순서는 그래픽에 대한 설명을 제공합니다. 연결 지향 설계에서 서버와 클라이언트 어플리케이션간 관계도 설명합니다. 각각의 흐름 세트에는 특정 API에서의 사용법 노트로의 링크가 포함되어 있습니다. 특정 API 사용에 대한 추가 정보가 필요하면 이런 링크를 사용할 수 있습니다.

예: 연결 지향 서버에서는 다음과 같은 함수 호출 순서를 사용합니다.

1. **socket()** 함수는 종료점을 표시하는 소켓 설명자를 리턴합니다. 이 명령문은 TCP 전송(SOCK_STREAM)을 사용하는 INET(인터넷 프로토콜) 주소 패밀리가 이 소켓에 사용된다는 것도 식별합니다.
2. **setsockopt()** 함수는 필수 대기 시간이 만기되기 전에 서버가 재시작될 때 로컬 주소를 재사용할 수 있도록 합니다.
3. 소켓 설명자가 작성된 후에 **bind()** 함수는 소켓에 고유한 이름을 제공합니다. 이 예에서 사용자는 s_addr을 0으로 설정하여 3005 포트를 지정하는 모든 IPv4 클라이언트에서 연결을 설정할 수 있도록 합니다.
4. **listen()**은 서버가 수신 클라이언트 연결을 허용할 수 있도록 합니다. 이 예에서는 백로그가 10으로 설정됩니다. 이것은 시스템이 수신 요구 거부를 시작하기 전에 10개의 수신 연결 요구를 대기행렬에 넣는다는 것을 의미합니다.
5. 서버에서는 수신 연결 요구를 승인하기 위해 **accept()** 함수를 사용합니다. **accept()** 호출은 수신 연결이 도착하길 기다리며 무기한 블록화됩니다.
6. **select()** 함수는 프로세스가 이벤트 발생을 기다릴 수 있도록 하며 이벤트가 발생할 때 프로세스를 깨울 수 있도록 합니다. 이 예에서는 자료를 읽을 수 있는 경우에만 프로세스에 통지합니다. 이 선택 호출에서는 30초의 시간종료가 사용됩니다.
7. **recv()** 함수는 클라이언트 어플리케이션으로부터 자료를 수신합니다. 이 예에서 우리는 클라이언트가 250 바이트의 자료를 송신한다는 것을 알고 있습니다. 이 사실을 알고 있기 때문에 SO_RCVLOWAT 소켓 옵션을 사용하여 250바이트의 자료가 모두 도착할 때까지 **recv()**가 깨어나지 않도록 지정할 수 있습니다.
8. **send()** 함수는 클라이언트로 자료를 다시 에코우합니다.
9. **close()** 함수는 열린 소켓 설명자를 닫습니다.

이벤트의 소켓 흐름: 연결 지향 클라이언트

예: 연결 지향 클라이언트에서는 다음과 같은 함수 호출 순서를 사용합니다.

1. **socket()** 함수는 종료점을 표시하는 소켓 설명자를 리턴합니다. 이 명령문은 TCP 전송(SOCK_STREAM)을 사용하는 INET(인터넷 프로토콜) 주소 패밀리가 이 소켓에 사용된다는 것도 식별합니다.
2. 클라이언트 프로그램 예에서 **inet_addr()** 함수로 전달된 서버 스트링이 점 십진 IP 주소가 아닌 경우에는 서버 호스트명으로 가정됩니다. 그런 경우 **gethostbyname()** 함수를 사용하여 서버의 IP 주소를 검색하십시오.
3. 소켓 설명자가 수신된 후에 서버와의 연결 설정에 **connect()** 함수가 사용됩니다.
4. **send()** 함수는 서버로 250바이트의 자료를 송신합니다.
5. **recv()** 함수는 서버가 250바이트의 자료를 다시 에코우하기를 기다립니다. 이 예에서 우리는 서버가 우리가 방금 송신한 것과 동일한 250바이트에 응답할 것이라는 사실을 알고 있습니다. 클라이언트 예에서 250 바이트의 자료가 별도 패킷에 도달하므로 250바이트가 모두 도달할 때까지 **recv()** 함수를 반복해서 사용합니다.
6. **close()** 함수는 열린 소켓 설명자를 닫습니다.

예: 연결 지향 서버

다음 코드 예는 연결 지향 서버를 작성하는 방법을 나타낸 것입니다. 이 예를 사용하여 사용자 고유의 소켓 서버 어플리케이션을 작성할 수 있습니다. 연결 지향 서버 설계는 가장 일반적인 소켓 어플리케이션 모델 중

하나입니다. 연결 지향 설계에서 서버 어플리케이션은 소켓을 작성하여 클라이언트 요구를 허용합니다. 코드 예를 사용하는 것에 대한 정보는 코드 면책사항을 참조하십시오.

```
/*********************************************************************/
/* This sample program provides a code for a connection-oriented server.  */
/*********************************************************************/

/*********************************************************************/
/* Header files needed for this sample program .                     */
/*********************************************************************/
#include <stdio.h>
#include <sys/time.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

/*********************************************************************/
/* Constants used by this program                                    */
/*********************************************************************/
#define SERVER_PORT     3005
#define BUFFER_LENGTH    250
#define FALSE            0

void main()
{
   /*********************************************************************/
  /* Variable and structure definitions. */
   /*********************************************************************/
   int    sd=-1, sd2=-1;
   int    rc, length, on=1;
   char buffer[BUFFER_LENGTH];
  fd_set read_fd;
  struct timeval timeout;
  struct sockaddr_in serveraddr;

   /*********************************************************************/
   /* A do/while(FALSE) loop is used to make error cleanup easier.  The   */
   /* close() of each of the socket descriptors is only done once at the  */
   /* very end of the program.                                          */
   /*********************************************************************/
   do
   {
      /*********************************************************************/
      /* The socket() function returns a socket descriptor representing   */
      /* an endpoint.  The statement also identifies that the INET        */
      /* (Internet Protocol) address family with the TCP transport        */
      /* (SOCK_STREAM) will be used for this socket.                       */
      /*********************************************************************/
  sd = socket(AF_INET, SOCK_STREAM, 0);
    if (sd < 0)
      {
    perror("socket() failed");
         break;
      }

      /*********************************************************************/
      /* The setsockopt() function is used to allow the local address to  */
      /* be reused when the server is restarted before the required wait  */
```

```
        /* time expires.                                                  */
        /*****************************************************************/
        rc = setsockopt(sd, SOL_SOCKET, SO_REUSEADDR, (char *)&on, sizeof(on));
        if (rc < 0)
        {
           perror("setsockopt(SO_REUSEADDR) failed");
           break;
        }

        /*****************************************************************/
        /* After the socket descriptor is created, a bind() function gets a */
        /* unique name for the socket.  In this example, the user sets the  */
        /* s_addr to zero, which allows connections to be established from  */
        /* any client that specifies port 3005.                          */
        /*****************************************************************/
        memset(&serveraddr, 0, sizeof(serveraddr));
    serveraddr.sin_family      = AF_INET;
           serveraddr.sin_port        = htons(SERVER_PORT);
    serveraddr.sin_addr.s_addr = htonl(INADDR_ANY);

        rc = bind(sd, (struct sockaddr *)&serveraddr, sizeof(serveraddr));
        if (rc < 0)
        {
      perror("bind() failed");
           break;
        }

        /*****************************************************************/
        /* The listen() function allows the server to accept incoming       */
        /* client connections.  In this example, the backlog is set to 10.  */
        /* This means that the system will queue 10 incoming connection     */
        /* requests before the system starts rejecting the incoming         */
        /* requests.                                                     */
        /*****************************************************************/
     rc = listen(sd, 10);
        if (rc< 0)
        {
      perror("listen() failed");
           break;
        }

    printf("Ready for client connect().\n");

        /*****************************************************************/
        /* The server uses the accept() function to accept an incoming      */
        /* connection request.  The accept() call will block indefinitely   */
        /* waiting for the incoming connection to arrive.                */
        /*****************************************************************/
        sd2 = accept(sd, NULL, NULL);
        if (sd2 < 0)
        {
        perror("accept() failed");
           break;
        }

        /*****************************************************************/
        /* The select() function allows the process to wait for an event to */
        /* occur and to wake up the process when the event occurs.  In this */
```

```
|       /* example, the system notifies the process only when data is       */
|       /* available to read.  A 30 second timeout is used on this select   */
|       /* call.                                                            */
|       /*******************************************************************/
|       timeout.tv_sec  = 30;
|   timeout.tv_usec = 0;
|
|   FD_ZERO(&read_fd);
|       FD_SET(sd2, &read_fd);
|
|       rc = select(sd2+1, &read_fd, NULL, NULL, &timeout);
|       if (rc < 0)
|       {
|     perror("select() failed");
|          break;
|        }
|
|       if (rc == 0)
|       {
|     printf("select() timed out.\n");
|          break;
|       }
|
|       /*******************************************************************/
|       /* In this example we know that the client will send 250 bytes of   */
|       /* data over.  Knowing this, we can use the SO_RCVLOWAT socket      */
|       /* option and specify that we don't want our recv() to wake up until*/
|       /* all 250 bytes of data have arrived.                              */
|       /*******************************************************************/
|       length = BUFFER_LENGTH;
|       rc = setsockopt(sd2, SOL_SOCKET, SO_RCVLOWAT,
|                                        (char *)&length, sizeof(length));
|       if (rc < 0)
|       {
|          perror("setsockopt(SO_RCVLOWAT) failed");
|          break;
|       }
|
|       /*******************************************************************/
|       /* Receive that 250 bytes data from the client                      */
|       /*******************************************************************/
|       rc = recv(sd2, buffer, sizeof(buffer), 0);
|       if (rc < 0)
|       {
|       perror("recv() failed");
|          break;
|       }
|
|       printf("%d bytes of data were received\n", rc);
|       if (rc == 0 ||
|           rc < sizeof(buffer))
|       {
|          printf("The client closed the connection before all of the\n");
|          printf("data was sent\n");
|          break;
|       }
|
|       /*******************************************************************/
```

```
    /* Echo the data back to the client              */
       /*******************************************************************/
       rc = send(sd2, buffer, sizeof(buffer), 0);
       if (rc < 0)
       {
       perror("send() failed");
          break;
       }

       /*******************************************************************/
       /* Program complete                                                */
       /*******************************************************************/

    } while (FALSE);

    /*********************************************************************/
    /* Close down any open socket descriptors                            */
    /*********************************************************************/
    if (sd != -1)
             close(sd);
    if (sd2 != -1)
  close(sd2);
}
```

예: 연결 지향 클라이언트

다음 예는 연결 지향 설계에서 소켓 클라이언트 프로그램을 작성하여 연결 지향 서버에 연결하는 방법을 보여 줍니다. 서비스 클라이언트(클라이언트 프로그램)는 서버 프로그램에게 서비스를 요청해야 합니다. 이 코드 예를 사용하여 사용자 고유의 클라이언트 어플리케이션을 작성할 수 있습니다. 코드 예를 사용하는 것에 대한 정보는 코드 면책사항을 참조하십시오.

```
/*********************************************************************/
/* This sample program provides a code for a connection-oriented client.  */
/*********************************************************************/

/*********************************************************************/
/* Header files needed for this sample program                       */
/*********************************************************************/
#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>

/*********************************************************************/
/* Constants used by this program                                    */
/*********************************************************************/
#define SERVER_PORT     3005
#define BUFFER_LENGTH    250
#define FALSE            0
#define SERVER_NAME     "ServerHostName"

/* Pass in 1 parameter which is either the */
/* address or host name of the server, or  */
/* set the server name in the #define      */
```

```
/* SERVER_NAME.                              */
void main(int argc, char *argv[])
{
   /*********************************************************************/
  /* Variable and structure definitions. */
   /*********************************************************************/
   int    sd=-1, rc, bytesReceived;
   char buffer[BUFFER_LENGTH];
   char   server[NETDB_MAX_HOST_NAME_LENGTH];
  struct sockaddr_in serveraddr;
  struct hostent *hostp;

   /*********************************************************************/
   /* A do/while(FALSE) loop is used to make error cleanup easier.  The   */
   /* close() of the socket descriptor is only done once at the very end  */
   /* of the program.                                                     */
   /*********************************************************************/
   do
   {
      /*******************************************************************/
      /* The socket() function returns a socket descriptor representing   */
      /* an endpoint.  The statement also identifies that the INET        */
      /* (Internet Protocol) address family with the TCP transport        */
      /* (SOCK_STREAM) will be used for this socket.                      */
      /*******************************************************************/
  sd = socket(AF_INET, SOCK_STREAM, 0);
    if (sd < 0)
      {
    perror("socket() failed");
         break;
      }

      /*******************************************************************/
      /* If an argument was passed in, use this as the server, otherwise  */
      /* use the #define that is located at the top of this program.      */
      /*******************************************************************/
  if (argc > 1)
         strcpy(server, argv[1]);
   else
         strcpy(server, SERVER_NAME);

      memset(&serveraddr, 0, sizeof(serveraddr));
  serveraddr.sin_family      = AF_INET;
         serveraddr.sin_port        = htons(SERVER_PORT);
         serveraddr.sin_addr.s_addr = inet_addr(server);
         if (serveraddr.sin_addr.s_addr == (unsigned long)INADDR_NONE)
      {
         /****************************************************************/
         /* The server string that was passed into the inet_addr()        */
         /* function was not a dotted decimal IP address.  It must        */
         /* therefore be the hostname of the server.  Use the             */
         /* gethostbyname() function to retrieve the IP address of the    */
                  /* server.                                    */
         /****************************************************************/

      hostp = gethostbyname(server);
      if (hostp == (struct hostent *)NULL)
         {
```

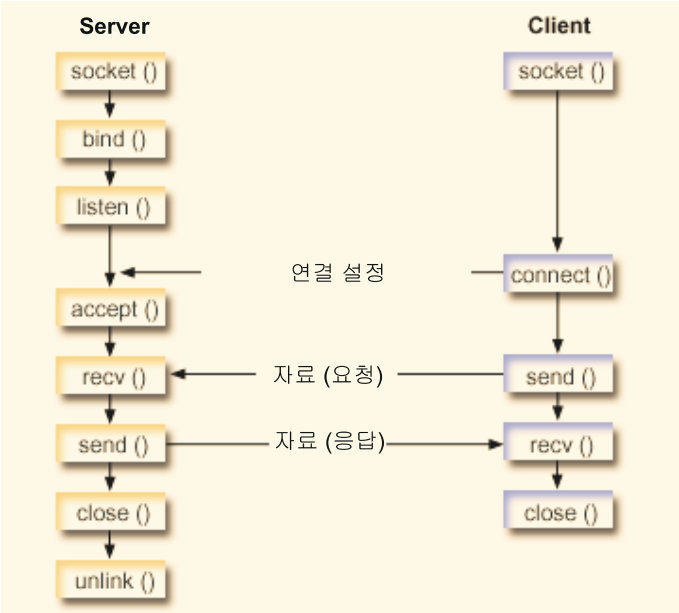
```
|              printf("Host not found --> ");
|            printf("h_errno = %d\n",h_errno);
|           break;
|           }
|
|         memcpy(&serveraddr.sin_addr,
|                hostp->h_addr,
|                sizeof(serveraddr.sin_addr));
|         }
|
|         /*********************************************************************/
|         /* Use the connect() function to establish a connection to the      */
|                     /* server.                                               */
|         /*********************************************************************/
|         rc = connect(sd, (struct sockaddr *)&serveraddr, sizeof(serveraddr));
|         if (rc < 0)
|         {
|       perror("connect() failed");
|            break;
|         }
|
|         /*********************************************************************/
|         /* Send 250 bytes of a's to the server                                */
|         /*********************************************************************/
|         memset(buffer, 'a', sizeof(buffer));
|         rc = send(sd, buffer, sizeof(buffer), 0);
|         if (rc < 0)
|         {
|         perror("send() failed");
|            break;
|         }
|
|         /*********************************************************************/
|         /* In this example we know that the server is going to respond with */
|         /* the same 250 bytes that we just sent.  Since we know that 250    */
|         /* bytes are going to be sent back to us, we could use the          */
|         /* SO_RCVLOWAT socket option and then issue a single recv() and     */
|         /* retrieve all of the data.                                         */
|         /*                                                                   */
|         /* The use of SO_RCVLOWAT is already illustrated in the server      */
|         /* side of this example, so we will do something different here.    */
|         /* The 250 bytes of the data may arrive in separate packets,        */
|         /* therefore we will issue recv() over and over again until all     */
|         /* 250 bytes have arrived.                                           */
|         /*********************************************************************/
|         bytesReceived = 0;
|         while (bytesReceived < BUFFER_LENGTH)
|         {
|            rc = recv(sd, & buffer[bytesReceived],
|                      BUFFER_LENGTH - bytesReceived, 0);
|         if (rc < 0)
|            {
|         perror("recv() failed");
|            break;
|            }
|           else if (rc == 0)
|            {
|               printf("The server closed the connection\n");
```

```
            break;
            }

            /*********************************************************************/
            /* Increment the number of bytes that have been received so far  */
            /*********************************************************************/
            bytesReceived += rc;
          }

      } while (FALSE);

      /*********************************************************************/
      /* Close down any open socket descriptors                            */
      /*********************************************************************/
      if (sd != -1)
               close(sd);
}
```

연결 소켓 작성

무접속 소켓은 연결을 설정하여 자료를 전송하지 않습니다. 대신 서버 어플리케이션이 클라이언트가 요구를 송신할 수 있는 이름을 지정합니다. 무접속 소켓은 TCP/IP 대신 사용자 데이터그램 프로토콜(UDP)을 사용합니다. 예: 무접속 서버 및 예: 무접속 클라이언트는 사용자 데이터그램 프로토콜(UDP)에 작성된 소켓 API를 설명합니다.

다음 그림은 무접속 소켓 설계를 위한 코드 예에서 사용된 소켓 API에서 클라이언트/서버 관계를 설명합니다.



이벤트의 소켓 흐름: 무접속 서버

다음 소켓 호출 순서는 그래픽에 대한 설명과 다음 프로그램 예를 제공합니다. 무접속 설계에서 서버와 클라이언트 어플리케이션간 관계도 설명합니다. 각각의 흐름 세트에는 특정 API에서의 사용법 노트로의 링크가 포함되어 있습니다. 특정 API 사용에 대한 추가 정보가 필요하면 이런 링크를 사용할 수 있습니다.

예: 연결 서버에서는 다음과 같은 함수 호출 순서를 사용합니다.

1. **socket()** 함수는 종료점을 표시하는 소켓 설명자를 리턴합니다. 이 명령문은 UDP 전송(SOCK_DGRAM)을 사용하는 INET(인터넷 프로토콜) 주소 패밀리가 이 소켓에 사용된다는 것도 식별합니다.
2. 소켓 설명자가 작성된 후에 **bind()** 함수는 소켓에 고유한 이름을 제공합니다. 이 예에서 사용자는 s_addr을 0으로 설정하여 3555 UDP 포트가 시스템의 모든 IPv4 주소에 바인드되도록 합니다.
3. 서버는 **recvfrom()** 함수를 사용하여 해당 자료를 수신합니다. **recvfrom()** 함수는 자료가 도착할 때까지 무기한 기다립니다.
4. **sendto()** 함수는 자료를 클라이언트로 다시 에코우합니다.
5. **close()** 함수는 열린 소켓 설명자를 종료합니다.

이벤트의 소켓 흐름: 무접속 클라이언트

예: 무접속 클라이언트에서는 다음 기능 호출 순서를 사용합니다.

1. **socket()** 함수는 종료점을 표시하는 소켓 설명자를 리턴합니다. 이 명령문은 UDP 전송(SOCK_DGRAM)을 사용하는 INET(인터넷 프로토콜) 주소 패밀리가 이 소켓에 사용된다는 것도 식별합니다.
2. 클라이언트 프로그램 예에서 inet_addr() 함수로 전달된 서버 스트링이 점 십진 IP 주소가 아닌 경우에는 서버 호스트명으로 가정됩니다. 그런 경우 **gethostbyname()** 함수를 사용하여 서버의 IP 주소를 검색하십시오.
3. 자료를 서버로 송신하려면 **sendto()** 함수를 사용하십시오.
4. 서버에서 자료를 다시 수신하려면 **recvfrom()** 함수를 사용하십시오.
5. **close()** 함수는 열린 소켓 설명자를 종료합니다.

예: 무접속 서버

이 예를 사용하여 무접속 서버 설계를 작성할 수 있습니다. 이 예에서는 UDP를 사용하여 무접속 소켓 서버 프로그램을 작성합니다. 코드 예를 사용하는 것에 대한 정보는 코드 면책사항을 참조하십시오.

```
/*********************************************************************/
/* This sample program provides a code for a connectionless server.  */
/*********************************************************************/

/*********************************************************************/
/* Header files needed for this sample program                       */
/*********************************************************************/
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

/*********************************************************************/
/* Constants used by this program                                    */
/*********************************************************************/
#define SERVER_PORT     3555
#define BUFFER_LENGTH    100
#define FALSE            0

void main()
{
```

```
 /*********************************************************************/
/* Variable and structure definitions. */
 /*********************************************************************/
 int    sd=-1, rc;
 char buffer[BUFFER_LENGTH];
struct sockaddr_in serveraddr;
 struct sockaddr_in clientaddr;
int clientaddrlen = sizeof(clientaddr);

 /*********************************************************************/
 /* A do/while(FALSE) loop is used to make error cleanup easier.  The   */
 /* close() of each of the socket descriptors is only done once at the  */
 /* very end of the program.                                            */
 /*********************************************************************/
 do
 {
    /*******************************************************************/
    /* The socket() function returns a socket descriptor representing   */
    /* an endpoint.  The statement also identifies that the INET        */
    /* (Internet Protocol) address family with the UDP transport        */
    /* (SOCK_DGRAM) will be used for this socket.                       */
    /*******************************************************************/
sd = socket(AF_INET, SOCK_DGRAM, 0);
  if (sd < 0)
    {
  perror("socket() failed");
       break;
    }

    /*******************************************************************/
    /* After the socket descriptor is created, a bind() function gets a */
    /* unique name for the socket.  In this example, the user sets the  */
    /* s_addr to zero, which means that the UDP port of 3555 will be    */
    /* bound to all IP addresses on the system.                         */
    /*******************************************************************/
    memset(&serveraddr, 0, sizeof(serveraddr));
serveraddr.sin_family      = AF_INET;
       serveraddr.sin_port        = htons(SERVER_PORT);
serveraddr.sin_addr.s_addr = htonl(INADDR_ANY);

    rc = bind(sd, (struct sockaddr *)&serveraddr, sizeof(serveraddr));
    if (rc < 0)
    {
  perror("bind() failed");
       break;
    }

    /*******************************************************************/
    /* The server uses the recvfrom() function to receive that data.    */
    /* The recvfrom() function waits indefinitely for data to arrive.   */
    /*******************************************************************/
    rc = recvfrom(sd, buffer, sizeof(buffer), 0,
            (struct sockaddr *)&clientaddr,
              &clientaddrlen);
    if (rc < 0)
    {
    perror("recvfrom() failed");
       break;
```

```
      }

      printf("server received the following: <%s>\n", buffer);
      printf("from port %d and address %s\n",
          ntohs(clientaddr.sin_port),
          inet_ntoa(clientaddr.sin_addr));

      /*******************************************************************/
   /* Echo the data back to the client              */
      /*******************************************************************/
      rc = sendto(sd, buffer, sizeof(buffer), 0,
              (struct sockaddr *)&clientaddr,
                  sizeof(clientaddr));
      if (rc < 0)
      {
      perror("sendto() failed");
         break;
      }

      /*******************************************************************/
      /* Program complete                                                */
      /*******************************************************************/

   } while (FALSE);

   /*******************************************************************/
   /* Close down any open socket descriptors                          */
   /*******************************************************************/
   if (sd != -1)
            close(sd);
}
```

예: 무접속 클라이언트

다음 예는 UDP를 사용하여 무접속 소켓 클라이언트 프로그램을 서버에 연결하는 방법을 보여줍니다. 코드 예를 사용하는 것에 대한 정보는 코드 면책사항을 참조하십시오.

```
/*********************************************************************/
/* This sample program provides a code for a connectionless client.  */
/*********************************************************************/

/*********************************************************************/
/* Header files needed for this sample program                       */
/*********************************************************************/
#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>

/*********************************************************************/
/* Constants used by this program                                    */
/*********************************************************************/
#define SERVER_PORT     3555
#define BUFFER_LENGTH    100
#define FALSE            0
```

```
#define SERVER_NAME     "ServerHostName"

/* Pass in 1 parameter which is either the */
/* address or host name of the server, or  */
/* set the server name in the #define      */
/* SERVER_NAME                             */
void main(int argc, char *argv[])
{
   /*********************************************************************/
  /* Variable and structure definitions. */
   /*********************************************************************/
  int sd, rc;
   char   server[NETDB_MAX_HOST_NAME_LENGTH];
   char buffer[BUFFER_LENGTH];
  struct hostent *hostp;
  struct sockaddr_in serveraddr;
  int serveraddrlen = sizeof(serveraddr);

   /*********************************************************************/
   /* A do/while(FALSE) loop is used to make error cleanup easier.  The   */
   /* close() of the socket descriptor is only done once at the very end  */
   /* of the program.                                                     */
   /*********************************************************************/
   do
   {
      /*******************************************************************/
      /* The socket() function returns a socket descriptor representing   */
      /* an endpoint.  The statement also identifies that the INET        */
      /* (Internet Protocol) address family with the UDP transport        */
      /* (SOCK_STREAM) will be used for this socket.                      */
      /*******************************************************************/
  sd = socket(AF_INET, SOCK_DGRAM, 0);
    if (sd < 0)
      {
    perror("socket() failed");
         break;
      }

      /*******************************************************************/
      /* If an argument was passed in, use this as the server, otherwise  */
      /* use the #define that is located at the top of this program.      */
      /*******************************************************************/
  if (argc > 1)
         strcpy(server, argv[1]);
   else
         strcpy(server, SERVER_NAME);

      memset(&serveraddr, 0, sizeof(serveraddr));
  serveraddr.sin_family      = AF_INET;
         serveraddr.sin_port        = htons(SERVER_PORT);
         serveraddr.sin_addr.s_addr = inet_addr(server);
         if (serveraddr.sin_addr.s_addr == (unsigned long)INADDR_NONE)
      {
         /*****************************************************************/
         /* The server string that was passed into the inet_addr()        */
         /* function was not a dotted decimal IP address.  It must        */
         /* therefore be the hostname of the server.  Use the             */
         /* gethostbyname() function to retrieve the IP address of the    */
```

```
            /* server.                                          */
      /*****************************************************************/
    hostp = gethostbyname(server);
    if (hostp == (struct hostent *)NULL)
       {
          printf("Host not found --> ");
        printf("h_errno = %d\n",h_errno);
       break;
       }

    memcpy(&serveraddr.sin_addr,
           hostp->h_addr,
           sizeof(serveraddr.sin_addr));
    }

    /*****************************************************************/
    /* Initialize the data block that is going to be sent to the server */
    /*****************************************************************/
    memset(buffer, 0, sizeof(buffer));
    strcpy(buffer, "A CLIENT REQUEST");

    /*****************************************************************/
    /* Use the sendto() function to send the data to the server.        */
    /*****************************************************************/
    rc = sendto(sd, buffer, sizeof(buffer), 0,
            (struct sockaddr *)&serveraddr,
            sizeof(serveraddr));
    if (rc < 0)
    {
    perror("sendto() failed");
       break;
    }

    /*****************************************************************/
    /* Use the recvfrom() function to receive the data back from the    */
            /* server.                                          */
    /*****************************************************************/
    rc = recvfrom(sd, buffer, sizeof(buffer), 0,
            (struct sockaddr *)&serveraddr,
                  & serveraddrlen);
    if (rc < 0)
    {
    perror("recvfrom() failed");
       break;
    }

    printf("client received the following: <%s>\n", buffer);
printf(" from port %d, from address %s\n",
        ntohs(serveraddr.sin_port),
        inet_ntoa(serveraddr.sin_addr));

    /*****************************************************************/
    /* Program complete                                              */
    /*****************************************************************/

 } while (FALSE);

 /*****************************************************************/
```

```
   /* Close down any open socket descriptors                            */
   /*********************************************************************/
   if (sd != -1)
            close(sd);
}
```

주소 패밀리를 사용하여 어플리케이션 설계

다음 주제에서는 각 소켓 주소 패밀리를 보여주는 샘플 프로그램을 제공합니다.

- AF_INET 주소 패밀리 사용
- AF_INET6 주소 패밀리 사용
- AF_UNIX 주소 패밀리 사용
- AF_TELEPHONY 주소 패밀리 사용
- AF_UNIX_CCSID 주소 패밀리 사용

AF_INET 주소 패밀리 사용

AF_INET 주소 패밀리 소켓은 연결 지향(SOCK_STREAM 유형)이거나 무접속(SOCK_DGRAM 유형)일 수 있습니다. 연결 지향 AF_INET 소켓은 전송 프로토콜로 TCP를 사용합니다. 무접속 AF_INET 소켓은 전송 프로토콜로 UDP를 사용합니다. AF_INET 정의역 소켓 작성시 소켓 프로그램에서 주소 패밀리에 AF_INET를 지정합니다. AF_INET 소켓은 SOCK_RAW 유형을 사용할 수도 있습니다. 이 유형이 설정되면 어플리케이션은 직접 IP층에 연결하여 TCP 또는 UDP 전송을 사용하지 않습니다.

AF_INET 주소 패밀리를 사용하기 위해 환경을 설정하는 것에 대한 세부사항은 소켓 프로그래밍의 전제조건을 참조하십시오.

AF_INET 주소 패밀리를 사용하는 샘플 프로그램은 예: 연결 지향 서버 및 예: 연결 지향 클라이언트를 참조하십시오.

AF_INET6 주소 패밀리 사용

AF_INET6 소켓은 인터넷 프로토콜 버전 6(IPv6) 128비트(16바이트) 주소 구조를 지원합니다. 프로그래머는 AF_INET6 주소 패밀리를 사용하여 IPv4나 IPv6 노드에 대해 또는 IPv6 노드에서만 클라이언트 요구를 허용하도록 어플리케이션을 작성할 수 있습니다.

AF_INET 소켓처럼 AF_INET6 소켓은 연결 지향(SOCK_STREAM 유형)이거나 무접속(SOCK_DGRAM 유형)일 수 있습니다. 연결 지향 AF_INET6 소켓은 전송 프로토콜로 TCP를 사용합니다. 무접속 AF_INET6 소켓은 전송 프로토콜로 UDP를 사용합니다. AF_INET6 정의역 소켓 작성시 소켓 프로그램에서 주소 패밀리에 AF_INET6을 지정합니다. AF_INET6 소켓은 SOCK_RAW 유형을 사용할 수도 있습니다. 이 유형이 설정되면 어플리케이션은 직접 IP층에 연결하여 TCP 또는 UDP 전송을 사용하지 않습니다. AF_INET6 주소 패밀리를 사용하기 위해 환경을 설정하는 것에 대한 세부사항은 소켓 프로그래밍의 전제조건을 참조하십시오.

IPv4 어플리케이션과의 IPv6 어플리케이션 호환성

AF_INET6 주소 패밀리를 사용하여 작성된 소켓 어플리케이션은 인터넷 프로토콜 버전 6(IPv6) 어플리케이션이 인터넷 프로토콜 버전 4(IPv4) 어플리케이션(AF_INET 주소 패밀리를 사용하는 어플리케이션)에 대한 작업을 할 수 있도록 합니다. 이 피처는 소켓 프로그래머가 IPv4-맵핑 IPv6 주소 형식을 사용할 수 있도록 합니다. 이 주소 형식은 IPv4 노드의 IPv4 주소가 IPv6 주소로 표시되도록 표시합니다. IPv4 주소는 IPv6 주소의 하위 32비트로 코드화되며 상위 96비트는 고정 접두부 0:0:0:0:0:FFFF를 보유합니다. 예를 들어, IPv4-맵핑 주소는 다음과 같습니다.

```
::FFFF:192.1.1.1
```

지정된 호스트에 IPv4 주소만 있는 경우 **getaddrinfo()** 함수로 이 주소를 자동 생성할 수 있습니다.

AF_INET6 소켓을 사용하는 어플리케이션을 작성하여 IPv4 노드로 TCP 연결을 열 수 있습니다. 이 타스크를 완료하기 위해 목적지의 IPv4 주소를 IPv4-맵핑 IPv6 주소로 코드화하고 **connect()** 또는 **sendto()** 호출시 sockaddr_in6 구조 내에서 해당 주소를 전달할 수 있습니다. 어플리케이션이 AF_INET6 소켓을 사용하여 IPv4 노드에서 TCP 연결을 허용할 때 IPv4 노드에서 UDP 패킷을 수신할 때 시스템은 이런 식으로 코드화된 sockaddr_in6 구조를 사용하여 **accept()**, **recvfrom()** 또는 **getpeername()** 호출시 어플리케이션에 피어 주소를 리턴합니다.

bind() 함수를 사용하면 어플리케이션이 UDP 패킷 및 TCP 연결의 소스 IP 주소를 선택할 수 있는 반면에 어플리케이션은 시스템이 소스 IP 주소를 선택하기 원하는 경우도 있습니다. 어플리케이션은 이런 목적으로 IPv4에서 INADDR_ANY 매크로를 사용하는 것과 유사한 방법으로 in6addr_any를 사용합니다. 이런 방법으로 바인딩하는 추가 피처는 AF_INET6 소켓이 IPv4 및 IPv6 노드 모두와 통신할 수 있도록 하는 것입니다. 예를 들어, in6addr_any로 바인드된 청취 소켓에서 **accept()**를 발행하는 어플리케이션은 IPv4 또는 IPv6 노드로부터 연결을 허용합니다. IPPROTO_IPV6 레벨 소켓 옵션 IPV6_V6ONLY를 사용하여 이 작동을 수정할 수 있습니다. 상호운영하고 있는 노드 유형을 알아야 하는 어플리케이션은 거의 없습니다. 그러나 노드 유형을 알아야 하는 어플리케이션의 경우 <netinet/in.h>에 정의된 IN6_IS_ADDR_V4MAPPED() 매크로가 제공됩니다.

IPv4와 IPv6를 보다 자세히 비교하고 싶으면 Information Center의 IPv4와 IPv6 비교를 참조하십시오. 이 정보는 두 프로토콜간 피처를 비교합니다.

AF_INET6 소켓이 IPv4 및 IPv6 노드 모두와 통신하는 상황에 대한 설명과 프로그램 예는 IPv4 및 IPv6 클라이언트를 허용하기 위한 어플리케이션 작성을 참조하십시오.

IPv6 제한사항

현재 IPv6에 대한 OS/400 지원은 V5R2에서 일부 기능으로 제한되어 있습니다. 다음 표는 이런 제한사항과 소켓 프로그래머에게 미치는 영향을 나열한 것입니다.

표 12. IPv6 제한사항 및 영향

제한사항	영향
IPv6은 프래그먼트화를 지원하지 않습니다.	AF_INET6(SOCK_DGRAM)은 인터페이스의 MTU에서 헤더 크기를 뺀 것보다 더 큰 데이터그램을 송신하려고 시도해서는 안됩니다.

표 12. *IPv6 제한사항 및 영향 (계속)*

IPv6 애니캐스트는 지원되지 않습니다.	anycast 주소에 연결하거나 애니캐스트 주소로 송신할 수 없습니다.
IPv6 멀티캐스트는 지원되지 않습니다.	멀티캐스트 데이터그램을 송/수신할 수 없습니다.
iSeries 호스트 표는 IPv6 주소를 지원하지 않습니다.	**getaddrinfo()** 및 **getnameinfo()** API는 호스트 표에서 IPv6 주소를 찾을 수 없습니다. 이런 API는 DNS에서만 주소를 찾습니다.
gethostbyname() 및 **gethostbyaddr()** API는 IPv4 주소 분석만 지원합니다.	IPv6 주소 분석이 필요한 경우 **getaddrinfo()** 및 **getnameinfo()** API를 사용하십시오.

AF_UNIX 주소 패밀리

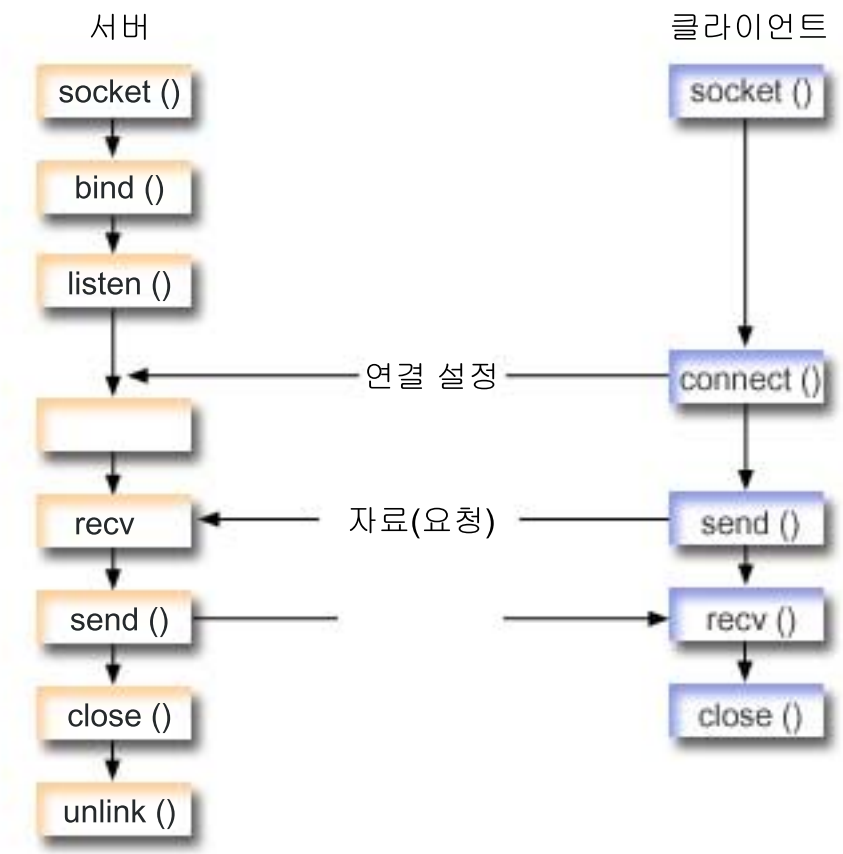
AF_UNIX 주소 패밀리(AF_UNIX 또는 AF_UNIX_CCSID 주소 패밀리를 사용하는 소켓)는 연결 지향(SOCK_STREAM 유형)이거나 무접속(SOCK_DGRAM 유형)일 수 있습니다. 두 프로세스를 연결하는 외부 통신 함수가 없으므로 두 유형 모두 신뢰할 수 있습니다.

UNIX 정의역 데이터그램 소켓은 UDP 데이터그램 소켓과는 다른 기능을 합니다. UDP 데이터그램 소켓에서 클라이언트 프로그램은 **bind()** 함수를 호출할 필요가 없습니다. 시스템이 사용되지 않는 포트 번호를 자동으로 할당하기 때문입니다. 그 다음, 서버는 데이터그램을 다시 해당 포트 번호로 송신합니다. 그러나 UNIX 정의역 데이터그램 소켓에서는 시스템이 자동으로 클라이언트에 대한 경로명을 할당하지 않습니다. 따라서 UNIX 정의역 데이터그램을 사용하는 모든 클라이언트 프로그램에서는 **bind()** 함수를 호출해야 합니다. 클라이언트의 **bind()**에 지정된 정확한 경로명이 서버에 전달됩니다. 따라서 클라이언트에서 상대 경로명(즉, /로 시작하여 완전히 규정되지 않은 경로명)을 지정하면 서버는 동일한 현재 디렉토리에서 실행되지 않는 한 클라이언트에 데이터그램을 송신할 수 없습니다.

어플리케이션이 이 주소 패밀리에 사용할 수 있는 경로명 예로는 /tmp/myserver 또는 servers/thatserver가 있습니다. servers/thatserver의 경우 완전히 규정되지 않은(/이 지정되지 않음) 경로명이 있습니다. 이것은 파일 시스템 계층 내의 항목 위치가 현재 작업 디렉토리에 따라 결정되어야 함을 의미합니다.

주: 파일 시스템의 경로명은 NLS-enabled입니다.

다음 그림은 AF_UNIX 주소 패밀리의 클라이언트/서버 관계를 설명합니다. AF_UNIX 주소 패밀리를 사용하기 위해 환경을 설정하는 것에 대한 세부사항은 소켓 프로그래밍의 전제조건을 참조하십시오.



이벤트의 소켓 흐름: AF_UNIX 주소 패밀리를 사용하는 서버 어플리케이션

예: AF_UNIX 주소 패밀리를 사용하는 서버 어플리케이션에서는 다음 기능 호출 순서를 사용합니다.

1. **socket()** 함수는 종료점을 표시하는 소켓 설명자를 리턴합니다. 이 명령문은 스트림 전송(SOCK_STREAM)을 사용하는 UNIX 주소 패밀리가 이 소켓에 사용된다는 것도 식별합니다. 이 함수는 종료점을 설명하는 소켓 설명자를 리턴합니다. **socketpair()** 함수를 사용하여 UNIX 소켓을 초기화할 수도 있습니다.

   AF_UNIX 또는 AF_UNIX_CCSID는 **socketpair()** 함수를 지원하기 위한 유일한 주소 패밀리입니다. **socketpair()** 함수는 명명되지 않고 연결된 2개의 소켓 설명자를 리턴합니다.
2. 소켓 설명자가 작성된 후에 **bind()** 함수는 소켓에 고유한 이름을 제공합니다.

   UNIX 정의역 소켓의 이름 공간은 경로명으로 구성됩니다. 소켓 프로그램이 **bind()** 함수를 호출할 때 파일 시스템 디렉토리에 항목이 작성됩니다. 경로명이 이미 존재하는 경우 **bind()**는 실패합니다. 따라서 UNIX 정의역 소켓 프로그램은 항상 종료시 **unlink()** 함수를 호출하여 디렉토리 항목을 제거해야 합니다.
3. **listen()**은 서버가 수신 클라이언트 연결을 허용할 수 있도록 합니다. 이 예에서는 백로그가 10으로 설정됩니다. 이것은 시스템이 수신 요구 거부를 시작하기 전에 10개의 수신 연결 요구를 대기행렬에 넣는다는 것을 의미합니다.
4. **recv()** 함수는 클라이언트 어플리케이션으로부터 자료를 수신합니다. 이 예에서 우리는 클라이언트가 250바이트의 자료를 송신한다는 것을 알고 있습니다. 이 사실을 알고 있기 때문에 SO_RCVLOWAT 소켓 옵션을 사용하여 250바이트의 자료가 모두 도착할 때까지 **recv()**가 깨어나지 않도록 지정할 수 있습니다.

5. **send()** 함수는 클라이언트로 자료를 다시 에코우합니다.
6. **close()** 함수는 열린 소켓 설명자를 닫습니다.
7. **unlink()** 함수는 파일 시스템에서 UNIX 경로명을 제거합니다.

이벤트의 소켓 흐름: AF_UNIX 주소 패밀리를 사용하는 클라이언트 어플리케이션

예: AF_UNIX 주소 패밀리를 사용하는 클라이언트 어플리케이션에서는 다음 기능 호출 순서를 사용합니다.

1. **socket()** 함수는 종료점을 표시하는 소켓 설명자를 리턴합니다. 이 명령문은 스트림 전송(SOCK_STREAM)을 사용하는 UNIX 주소 패밀리가 이 소켓에 사용된다는 것도 식별합니다. 이 함수는 종료점을 설명하는 소켓 설명자를 리턴합니다. **socketpair()** 함수를 사용하여 UNIX 소켓을 초기화할 수도 있습니다.

   AF_UNIX 또는 AF_UNIX_CCSID는 **socketpair()** 함수를 지원하기 위한 유일한 주소 패밀리입니다. **socketpair()** 함수는 명명되지 않고 연결된 2개의 소켓 설명자를 리턴합니다.
2. 소켓 설명자가 수신된 후에 서버와의 연결 설정에 **connect()** 함수가 사용됩니다.
3. **send()** 함수는 SO_RCVLOWAT 소켓 옵션을 사용하여 서버 어플리케이션에서 지정된 250바이트의 자료를 송신합니다.
4. 250바이트의 자료가 모두 도달할 때까지 **recv()** 함수는 루프됩니다.
5. **close()** 함수는 열린 소켓 설명자를 닫습니다.

예: AF_UNIX 주소 패밀리를 사용하는 서버 어플리케이션

이 예에서는 AF_UNIX 주소 패밀리에 샘플 서버를 제공합니다. 경로명 구조를 사용하여 서버 어플리케이션을 식별하는 경우를 제외하고 AF_UNIX 주소 패밀리는 다른 주소 패밀리와 동일한 다수의 소켓 호출을 사용합니다. 다음 샘플 프로그램에서는 AF_UNIX 주소 패밀리를 사용합니다. 코드 예를 사용하는 것에 대한 정보는 코드 면책사항을 참조하십시오.

```
/*****************************************************************************/
/* This sample program provides code for aserver application that uses     */
/* AF_UNIX address family                                                    */
/*****************************************************************************/

/*****************************************************************************/
/* Header files needed for this sample program                               */
/*****************************************************************************/
#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/un.h>

/*****************************************************************************/
/* Constants used by this program                                            */
/*****************************************************************************/
#define SERVER_PATH     "/tmp/server"
#define BUFFER_LENGTH    250
#define FALSE            0

void main()
{
   /***************************************************************************/
  /* Variable and structure definitions. */
```

```
    /*****************************************************************/
    int    sd=-1, sd2=-1;
    int    rc, length;
    char buffer[BUFFER_LENGTH];
    struct sockaddr_un serveraddr;

    /*****************************************************************/
    /* A do/while(FALSE) loop is used to make error cleanup easier.  The   */
    /* close() of each of the socket descriptors is only done once at the  */
    /* very end of the program.                                            */
    /*****************************************************************/
    do
    {
       /**************************************************************/
       /* The socket() function returns a socket descriptor representing   */
       /* an endpoint.  The statement also identifies that the UNIX        */
       /* address family with the stream transport (SOCK_STREAM) will be   */
       /* used for this socket.                                            */
       /**************************************************************/
       sd = socket(AF_UNIX, SOCK_STREAM, 0);
     if (sd < 0)
       {
     perror("socket() failed");
          break;
       }

       /**************************************************************/
       /* After the socket descriptor is created, a bind() function gets a */
       /* unique name for the socket.                                      */
       /**************************************************************/
       memset(&serveraddr, 0, sizeof(serveraddr));
    serveraddr.sun_family        = AF_UNIX;
          strcpy(serveraddr.sun_path, SERVER_PATH);

       rc = bind(sd, (struct sockaddr *)&serveraddr, SUN_LEN(&serveraddr));
       if (rc < 0)
       {
     perror("bind() failed");
          break;
       }

       /**************************************************************/
       /* The listen() function allows the server to accept incoming       */
       /* client connections.  In this example, the backlog is set to 10.  */
       /* This means that the system will queue 10 incoming connection     */
       /* requests before the system starts rejecting the incoming         */
       /* requests.                                                        */
       /**************************************************************/
    rc = listen(sd, 10);
       if (rc< 0)
       {
     perror("listen() failed");
          break;
       }

   printf("Ready for client connect().\n");

       /**************************************************************/
```

```
      /* The server uses the accept() function to accept an incoming      */
      /* connection request.  The accept() call will block indefinitely   */
      /* waiting for the incoming connection to arrive.                   */
      /********************************************************************/
      sd2 = accept(sd, NULL, NULL);
      if (sd2 < 0)
      {
      perror("accept() failed");
         break;
      }

      /********************************************************************/
      /* In this example we know that the client will send 250 bytes of   */
      /* data over.  Knowing this, we can use the SO_RCVLOWAT socket      */
      /* option and specify that we don't want our recv() to wake up      */
      /* until all 250 bytes of data have arrived.                        */
      /********************************************************************/
      length = BUFFER_LENGTH;
      rc = setsockopt(sd2, SOL_SOCKET, SO_RCVLOWAT,
                                        (char *)&length, sizeof(length));
      if (rc < 0)
      {
      }

      printf("%d bytes of data were received\n", rc);
      if (rc == 0 ||
          rc < sizeof(buffer))
      {
         printf("The client closed the connection before all of the\n");
         printf("data was sent\n");
         break;
      }

      /********************************************************************/
    /* Echo the data back to the client              */
      /********************************************************************/
      rc = send(sd2, buffer, sizeof(buffer), 0);
      if (rc < 0)
      {
      perror("send() failed");
         break;
      }

      /********************************************************************/
      /* Program complete                                                 */
      /********************************************************************/

    } while (FALSE);

    /*********************************************************************/
    /* Close down any open socket descriptors                            */
    /*********************************************************************/
    if (sd != -1)
             close(sd);

    if (sd2 != -1)
   close(sd2);
```

```
    /*********************************************************************/
    /* Remove the UNIX path name from the file system                    */
    /*********************************************************************/
    unlink(SERVER_PATH);
}
```

예: AF_UNIX 주소 패밀리를 사용하는 클라이언트 어플리케이션

이 예에서는 AF_UNIX 주소 패밀리에 샘플 클라이언트 어플리케이션을 제공합니다. 경로명 구조를 사용하여 서버 어플리케이션을 식별하는 경우를 제외하고 AF_UNIX 주소 패밀리는 다른 주소 패밀리와 동일한 다수의 소켓 호출을 사용합니다. 다음 샘플 프로그램에서는 AF_UNIX 주소 패밀리를 사용하여 서버와의 클라이언트 연결을 작성합니다. 코드 예를 사용하는 것에 대한 정보는 코드 면책사항를 참조하십시오.

```
/*********************************************************************/
/* This sample program provides code for a client application that uses     */
/* AF_UNIX address family                                            */
/*********************************************************************/
/*********************************************************************/
/* Header files needed for this sample program                       */
/*********************************************************************/
#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/un.h>

/*********************************************************************/
/* Constants used by this program                                    */
/*********************************************************************/
#define SERVER_PATH     "/tmp/server"
#define BUFFER_LENGTH    250
#define FALSE            0

/* Pass in 1 parameter which is either the */
/* path name of the server as a UNICODE    */
/* string, or set the server path in the   */
/* #define SERVER_PATH which is a CCSID    */
/* 500 string.                             */
void main(int argc, char *argv[])
{
   /*********************************************************************/
  /* Variable and structure definitions. */
   /*********************************************************************/
   int    sd=-1, rc, bytesReceived;
   char buffer[BUFFER_LENGTH];
   struct sockaddr_un serveraddr;

   /*********************************************************************/
   /* A do/while(FALSE) loop is used to make error cleanup easier.  The   */
   /* close() of the socket descriptor is only done once at the very end  */
   /* of the program.                                                   */
   /*********************************************************************/
   do
   {
      /*******************************************************************/
      /* The socket() function returns a socket descriptor representing   */
      /* an endpoint.  The statement also identifies that the UNIX_CCSID  */
      /* address family with the stream transport (SOCK_STREAM) will be   */
```

```
|       /* used for this socket.                                          */
|       /*********************************************************************/
|       sd = socket(AF_UNIX_CCSID, SOCK_STREAM, 0);
|     if (sd < 0)
|       {
|     perror("socket() failed");
|          break;
|       }
|
|       /*********************************************************************/
|       /* If an argument was passed in, use this as the server, otherwise  */
|       /* use the #define that is located at the top of this program.      */
|       /*********************************************************************/
|       memset(&serveraddr, 0, sizeof(serveraddr));
|    serveraddr.sun_family        = AF_UNIX;
|   if (argc > 1)
|          strcpy(serveraddr.sun_path, argv[1]);
|    else
|          strcpy(serveraddr.sun_path, SERVER_PATH);
|
|       /*********************************************************************/
|       /* Use the connect() function to establish a connection to the      */
|                   /* server.                                         */
|       /*********************************************************************/
|       rc = connect(sd, (struct sockaddr *)&serveraddr, SUN_LEN(&serveraddr));
|       if (rc < 0)
|       {
|     perror("connect() failed");
|          break;
|       }
|
|       /*********************************************************************/
|       /* Send 250 bytes of a's to the server                               */
|       /*********************************************************************/
|       memset(buffer, 'a', sizeof(buffer));
|       rc = send(sd, buffer, sizeof(buffer), 0);
|       if (rc < 0)
|       {
|       perror("send() failed");
|          break;
|       }
|
|       /*********************************************************************/
|       /* In this example we know that the server is going to respond with */
|       /* the same 250 bytes that we just sent.  Since we know that 250    */
|       /* bytes are going to be sent back to us, we could use the          */
|       /* SO_RCVLOWAT socket option and then issue a single recv() and     */
|       /* retrieve all of the data.                                         */
|       /*                                                                   */
|       /* The use of SO_RCVLOWAT is already illustrated in the server      */
|       /* side of this example, so we will do something different here.    */
|       /* The 250 bytes of the data may arrive in separate packets,        */
|       /* therefore we will issue recv() over and over again until all     */
|       /* 250 bytes have arrived.                                           */
|       /*********************************************************************/
|       bytesReceived = 0;
|       while (bytesReceived < BUFFER_LENGTH)
|       {
```

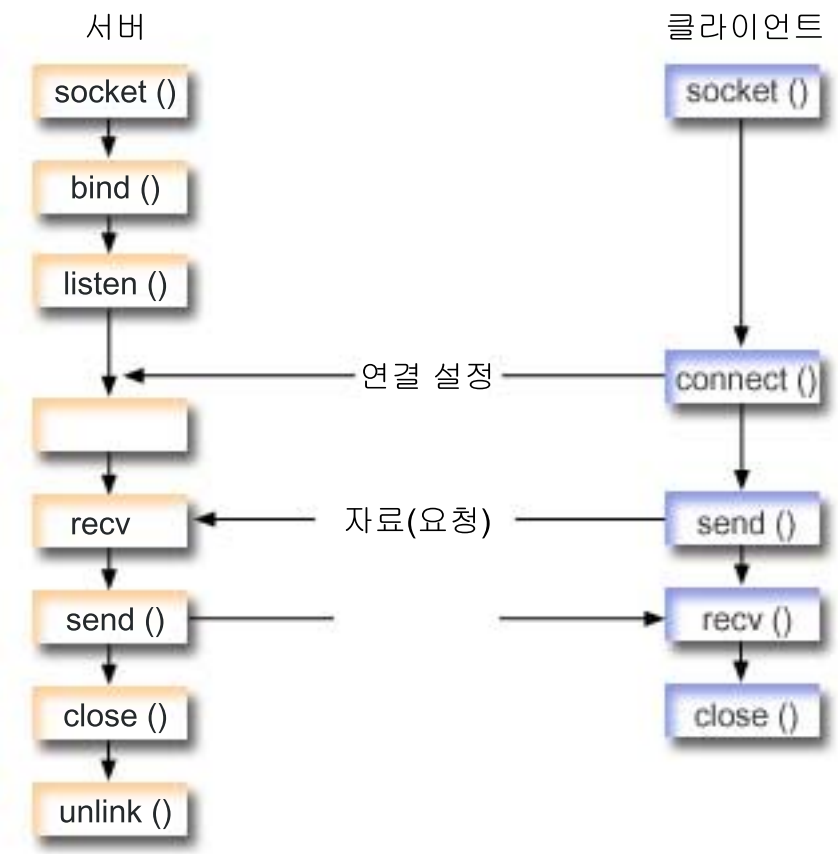
```
          rc = recv(sd, & buffer[bytesReceived],
                    BUFFER_LENGTH - bytesReceived, 0);
       if (rc < 0)
          {
       perror("recv() failed");
          break;
          }
         else if (rc == 0)
          {
             printf("The server closed the connection\n");
          break;
          }

          /*****************************************************************/
          /* Increment the number of bytes that have been received so far  */
          /*****************************************************************/
          bytesReceived += rc;
       }

   } while (FALSE);

   /*********************************************************************/
   /* Close down any open socket descriptors                            */
   /*********************************************************************/
   if (sd != -1)
            close(sd);
}
```

AF_UNIX_CCSID 주소 패밀리 사용

AF_UNIX_CCSID 주소 패밀리 소켓에는 AF_UNIX 주소 패밀리 소켓과 동일한 스펙이 있습니다. 이들은 연결 지향 또는 무접속에 사용될 수 있으며 동일한 시스템에서 통신을 제공합니다. 자세한 내용은 AF_UNIX 주소 패밀리 사용을 참조하십시오.

AF_UNIX_CCSID 소켓 어플리케이션에 대해 작업하기 전에, 출력 형식을 판별하려면 **Qlg_Path_Name_T** 구조에 익숙해 있어야 합니다. 자세한 내용은 Information Center의 API 참조 정보에서 경로명 구조를 참조하십시오.

accept(), **getsockname()**, **getpeername()**, **recvfrom()** 및 **recvmsg()**에서 리턴된 출력 주소 구조에 대한 작업시, 어플리케이션에서는 해당 형식을 판별할 소켓 주소 구조(sockaddr_unc)를 검사해야 합니다. sunc_format 및 sunc_qlg 필드는 경로명의 출력 형식을 판별합니다. 그러나 소켓은 입력 주소에서 사용된 어플리케이션과 동일한 출력의 값을 사용합니다.



이벤트의 소켓 흐름: AF_UNIX_CCSID 주소 패밀리를 사용하는 서버 어플리케이션

예: AF_UNIX_CCSID 주소 패밀리를 사용하는 서버 어플리케이션에서는 다음 기능 호출 순서를 사용합니다.

1. **socket()** 함수는 종료점을 표시하는 소켓 설명자를 리턴합니다. 이 명령문은 스트림 전송(SOCK_STREAM)을 사용하는 UNIX_CCSID 주소 패밀리가 이 소켓에 사용된다는 것도 식별합니다. **socketpair()** 함수를 사용하여 UNIX 소켓을 초기화할 수도 있습니다.

   AF_UNIX 또는 AF_UNIX_CCSID는 **socketpair()** 함수를 지원하기 위한 유일한 주소 패밀리입니다. **socketpair()** 함수는 명명되지 않고 연결된 2개의 소켓 설명자를 리턴합니다.
2. 소켓 설명자가 작성된 후에 **bind()** 함수는 소켓에 고유한 이름을 제공합니다.

   UNIX 정의역 소켓의 이름 공간은 경로명으로 구성됩니다. 소켓 프로그램이 **bind()** 함수를 호출할 때 파일 시스템 디렉토리에 항목이 작성됩니다. 경로명이 이미 존재하는 경우 **bind()**는 실패합니다. 따라서 UNIX 정의역 소켓 프로그램은 항상 종료시 **unlink()** 함수를 호출하여 디렉토리 항목을 제거해야 합니다.
3. **listen()**은 서버가 수신 클라이언트 연결을 허용할 수 있도록 합니다. 이 예에서는 백로그가 10으로 설정됩니다. 이것은 시스템이 수신 요구 거부를 시작하기 전에 10개의 수신 연결 요구를 대기행렬에 넣는다는 것을 의미합니다.
4. 서버에서는 수신 연결 요구를 승인하기 위해 **accept()** 함수를 사용합니다. **accept()** 호출은 수신 연결이 도착하길 기다리며 무기한 블록화됩니다.

5. **recv()** 함수는 클라이언트 어플리케이션으로부터 자료를 수신합니다. 이 예에서 우리는 클라이언트가 250 바이트의 자료를 송신한다는 것을 알고 있습니다. 이 사실을 알고 있기 때문에 SO_RCVLOWAT 소켓 옵션을 사용하여 250바이트의 자료가 모두 도착할 때까지 **recv()**가 깨어나지 않도록 지정할 수 있습니다.
6. **send()** 함수는 클라이언트로 자료를 다시 에코우합니다.
7. **close()** 함수는 열린 소켓 설명자를 닫습니다.
8. **unlink()** 함수는 파일 시스템에서 UNIX 경로명을 제거합니다.

이벤트의 소켓 흐름: AF_UNIX_CCSID 주소 패밀리를 사용하는 클라이언트 어플리케이션

예: AF_UNIX_CCSID 주소 패밀리를 사용하는 클라이언트 어플리케이션에서는 다음 기능 호출 순서를 사용합니다.

1. **socket()** 함수는 종료점을 표시하는 소켓 설명자를 리턴합니다. 이 명령문은 스트림 전송(SOCK_STREAM)을 사용하는 UNIX 주소 패밀리가 이 소켓에 사용된다는 것도 식별합니다. 이 함수는 종료점을 설명하는 소켓 설명자를 리턴합니다. **socketpair()** 함수를 사용하여 UNIX 소켓을 초기화할 수도 있습니다.

   AF_UNIX 또는 AF_UNIX_CCSID는 **socketpair()** 함수를 지원하기 위한 유일한 주소 패밀리입니다. **socketpair()** 함수는 명명되지 않고 연결된 2개의 소켓 설명자를 리턴합니다.
2. 소켓 설명자가 수신된 후에 서버와의 연결 설정에 **connect()** 함수가 사용됩니다.
3. **send()** 함수는 SO_RCVLOWAT 소켓 옵션을 사용하여 서버 어플리케이션에서 지정된 250바이트의 자료를 송신합니다.
4. 250바이트의 자료가 모두 도달할 때까지 **recv()** 함수는 루프됩니다.
5. **close()** 함수는 열린 소켓 설명자를 닫습니다.

예: AF_UNIX_CCSID 주소 패밀리를 사용하는 서버 어플리케이션

다음 샘플 프로그램에서는 AF_UNIX_CCSID 주소 패밀리를 사용합니다. 코드 예를 사용하는 것에 대한 정보는 코드 면책사항을 참조하십시오.

```
/*****************************************************************************/
/* This sample program provides code for a server application for            */
/* AF_UNIX_CCSID address family.                                             */
/*****************************************************************************/

/*****************************************************************************/
/* Header files needed for this sample program                               */
/*****************************************************************************/
#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/unc.h>

/*****************************************************************************/
/* Constants used by this program                                            */
/*****************************************************************************/
#define SERVER_PATH     "/tmp/server"
#define BUFFER_LENGTH    250
#define FALSE            0
```

```
void main()
{
   /*********************************************************************/
  /* Variable and structure definitions. */
   /*********************************************************************/
   int    sd=-1, sd2=-1;
   int    rc, length;
   char buffer[BUFFER_LENGTH];
   struct sockaddr_unc serveraddr;

   /*********************************************************************/
   /* A do/while(FALSE) loop is used to make error cleanup easier.  The   */
   /* close() of each of the socket descriptors is only done once at the  */
   /* very end of the program.                                            */
   /*********************************************************************/
   do
   {
      /*******************************************************************/
      /* The socket() function returns a socket descriptor representing   */
      /* an endpoint.  The statement also identifies that the UNIX_CCSID  */
      /* address family with the stream transport (SOCK_STREAM) will be   */
      /* used for this socket.                                            */
      /*******************************************************************/
      sd = socket(AF_UNIX_CCSID, SOCK_STREAM, 0);
    if (sd < 0)
      {
    perror("socket() failed");
         break;
      }

      /*******************************************************************/
      /* After the socket descriptor is created, a bind() function gets a */
      /* unique name for the socket.                                      */
      /*******************************************************************/
      memset(&serveraddr, 0, sizeof(serveraddr));
   serveraddr.sunc_family          = AF_UNIX_CCSID;
      serveraddr.sunc_format          = SO_UNC_USE_QLG;
      serveraddr.sunc_qlg.CCSID       = 500;
   serveraddr.sunc_qlg.Path_Type   = QLG_PTR_SINGLE;
         serveraddr.sunc_qlg.Path_Length = strlen(SERVER_PATH);
      serveraddr.sunc_path.p_unix     = SERVER_PATH;

      rc = bind(sd, (struct sockaddr *)&serveraddr, sizeof(serveraddr));
      if (rc < 0)
      {
    perror("bind() failed");
         break;
      }

      /*******************************************************************/
      /* The listen() function allows the server to accept incoming       */
      /* client connections.  In this example, the backlog is set to 10.  */
      /* This means that the system will queue 10 incoming connection     */
      /* requests before the system starts rejecting the incoming         */
      /* requests.                                                        */
      /*******************************************************************/
   rc = listen(sd, 10);
      if (rc< 0)
```

```
    {
  perror("listen() failed");
       break;
    }

printf("Ready for client connect().\n");

    /*********************************************************************/
    /* The server uses the accept() function to accept an incoming      */
    /* connection request.  The accept() call will block indefinitely   */
    /* waiting for the incoming connection to arrive.                    */
    /*********************************************************************/
    sd2 = accept(sd, NULL, NULL);
    if (sd2 < 0)
    {
    perror("accept() failed");
       break;
    }

    /*********************************************************************/
    /* In this example we know that the client will send 250 bytes of    */
    /* data over.  Knowing this, we can use the SO_RCVLOWAT socket       */
    /* option and specify that we don't want our recv() to wake up       */
    /* until all 250 bytes of data have arrived.                         */
    /*********************************************************************/
    length = BUFFER_LENGTH;
    rc = setsockopt(sd2, SOL_SOCKET, SO_RCVLOWAT,
                                       (char *)&length, sizeof(length));
    if (rc < 0)
    {
       perror("setsockopt(SO_RCVLOWAT) failed");
       break;
    }


    /*********************************************************************/
    /* Receive that 250 bytes data from the client                       */
    /*********************************************************************/
    rc = recv(sd2, buffer, sizeof(buffer), 0);
    if (rc < 0)
    {
    perror("recv() failed");
       break;
    }

    printf("%d bytes of data were received\n", rc);
    if (rc == 0 ||
        rc < sizeof(buffer))
    {
       printf("The client closed the connection before all of the\n");
       printf("data was sent\n");
       break;
    }

    /*********************************************************************/
 /* Echo the data back to the client              */
    /*********************************************************************/
    rc = send(sd2, buffer, sizeof(buffer), 0);
```

```
      if (rc < 0)
      {
      perror("send() failed");
         break;
      }

      /*******************************************************************/
      /* Program complete                                                */
      /*******************************************************************/

   } while (FALSE);

   /*********************************************************************/
   /* Close down any open socket descriptors                            */
   /*********************************************************************/
   if (sd != -1)
            close(sd);

   if (sd2 != -1)
  close(sd2);

   /*********************************************************************/
   /* Remove the UNIX path name from the file system                    */
   /*********************************************************************/
   unlink(SERVER_PATH);
}
```

예: AF_UNIX_CCSID 주소 패밀리를 사용하는 클라이언트 어플리케이션

다음 샘플 프로그램에서는 AF_UNIX_CCSID 주소 패밀리를 사용합니다. 코드 예를 사용하는 것에 대한 정보는 코드 면책사항을 참조하십시오.

```
/*********************************************************************/
/* This sample program provides code for a client application for    */
/* AF_UNIX_CCSID address family.                                     */
/*********************************************************************/

/*********************************************************************/
/* Header files needed for this sample program                       */
/*********************************************************************/
#include <stdio.h>
#include <string.h>
#include <wcstr.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/unc.h>

/*********************************************************************/
/* Constants used by this program                                    */
/*********************************************************************/
#define SERVER_PATH     "/tmp/server"
#define BUFFER_LENGTH    250
#define FALSE            0

/* Pass in 1 parameter which is either the */
/* path name of the server as a UNICODE    */
/* string, or set the server path in the   */
/* #define SERVER_PATH which is a CCSID    */
/* 500 string.                             */
```

```
void main(int argc, char *argv[])
{
   /*********************************************************************/
  /* Variable and structure definitions. */
   /*********************************************************************/
   int    sd=-1, rc, bytesReceived;
   char buffer[BUFFER_LENGTH];
   struct sockaddr_unc serveraddr;

   /*********************************************************************/
   /* A do/while(FALSE) loop is used to make error cleanup easier.  The   */
   /* close() of the socket descriptor is only done once at the very end  */
   /* of the program.                                                     */
   /*********************************************************************/
   do
   {
      /*******************************************************************/
      /* The socket() function returns a socket descriptor representing   */
      /* an endpoint.  The statement also identifies that the UNIX_CCSID  */
      /* address family with the stream transport (SOCK_STREAM) will be   */
      /* used for this socket.                                            */
      /*******************************************************************/
      sd = socket(AF_UNIX_CCSID, SOCK_STREAM, 0);
    if (sd < 0)
      {
    perror("socket() failed");
         break;
      }

      /*******************************************************************/
      /* If an argument was passed in, use this as the server, otherwise  */
      /* use the #define that is located at the top of this program.      */
      /*******************************************************************/
      memset(&serveraddr, 0, sizeof(serveraddr));
   serveraddr.sunc_family          = AF_UNIX_CCSID;
  if (argc > 1)
      {
         /* The argument is a UNICODE path name.  Use the default format  */
      serveraddr.sunc_format          = SO_UNC_DEFAULT;
         wcscpy(serveraddr.sunc_path.wide, (wchar_t *) argv[1]);
      }
   else
      {
         /* The local #define is CCSID 500.  Set the Qlg_Path_Name to use */
         /* the character format                                          */
      serveraddr.sunc_format          = SO_UNC_USE_QLG;
      serveraddr.sunc_qlg.CCSID       = 500;
      serveraddr.sunc_qlg.Path_Type   = QLG_CHAR_SINGLE;
         serveraddr.sunc_qlg.Path_Length = strlen(SERVER_PATH);
         strcpy((char *)&serveraddr.sunc_path, SERVER_PATH);
      }
      /*******************************************************************/
      /* Use the connect() function to establish a connection to the      */
                /* server.                                              */
      /*******************************************************************/
      rc = connect(sd, (struct sockaddr *)&serveraddr, sizeof(serveraddr));
      if (rc < 0)
      {
```

```
    perror("connect() failed");
         break;
      }

      /*******************************************************************/
      /* Send 250 bytes of a's to the server                             */
      /*******************************************************************/
      memset(buffer, 'a', sizeof(buffer));
      rc = send(sd, buffer, sizeof(buffer), 0);
      if (rc < 0)
      {
      perror("send() failed");
         break;
      }

      /*******************************************************************/
      /* In this example we know that the server is going to respond with */
      /* the same 250 bytes that we just sent.  Since we know that 250    */
      /* bytes are going to be sent back to us, we could use the          */
      /* SO_RCVLOWAT socket option and then issue a single recv() and     */
      /* retrieve all of the data.                                        */
      /*                                                                  */
      /* The use of SO_RCVLOWAT is already illustrated in the server      */
      /* side of this example, so we will do something different here.    */
      /* The 250 bytes of the data may arrive in separate packets,        */
      /* therefore we will issue recv() over and over again until all     */
      /* 250 bytes have arrived.                                          */
      /*******************************************************************/
      bytesReceived = 0;
      while (bytesReceived < BUFFER_LENGTH)
      {
         rc = recv(sd, & buffer[bytesReceived],
                   BUFFER_LENGTH - bytesReceived, 0);
      if (rc < 0)
         {
      perror("recv() failed");
         break;
         }
        else if (rc == 0)
         {
            printf("The server closed the connection\n");
         break;
         }

         /*****************************************************************/
         /* Increment the number of bytes that have been received so far  */
         /*****************************************************************/
         bytesReceived += rc;
      }

   } while (FALSE);

   /*********************************************************************/
   /* Close down any open socket descriptors                            */
   /*********************************************************************/
   if (sd != -1)
            close(sd);
}
```

AF_TELEPHONY 주소 패밀리 사용

전화 통신 주소 패밀리(AF_TELEPHONY 주소 패밀리를 사용하는 소켓)를 사용하여 사용자는 표준 소켓 API를 사용하여 접속된 ISDN 전화 네트워크를 통해 전화 호출을 시작(다이얼)하고 완료(응답)할 수 있습니다. 이 정의역에서 연결의 종료점을 형성하는 소켓은 실제로 전화 호출에서 피호출(수동 종료점) 및 호출(능동 종료점) 상대방이라고 합니다. AF_TELEPHONY 주소는 sockaddr_tel 주소 구조에 포함된 최대 40자리수(0 - 9)로 구성되는 전화 번호입니다.

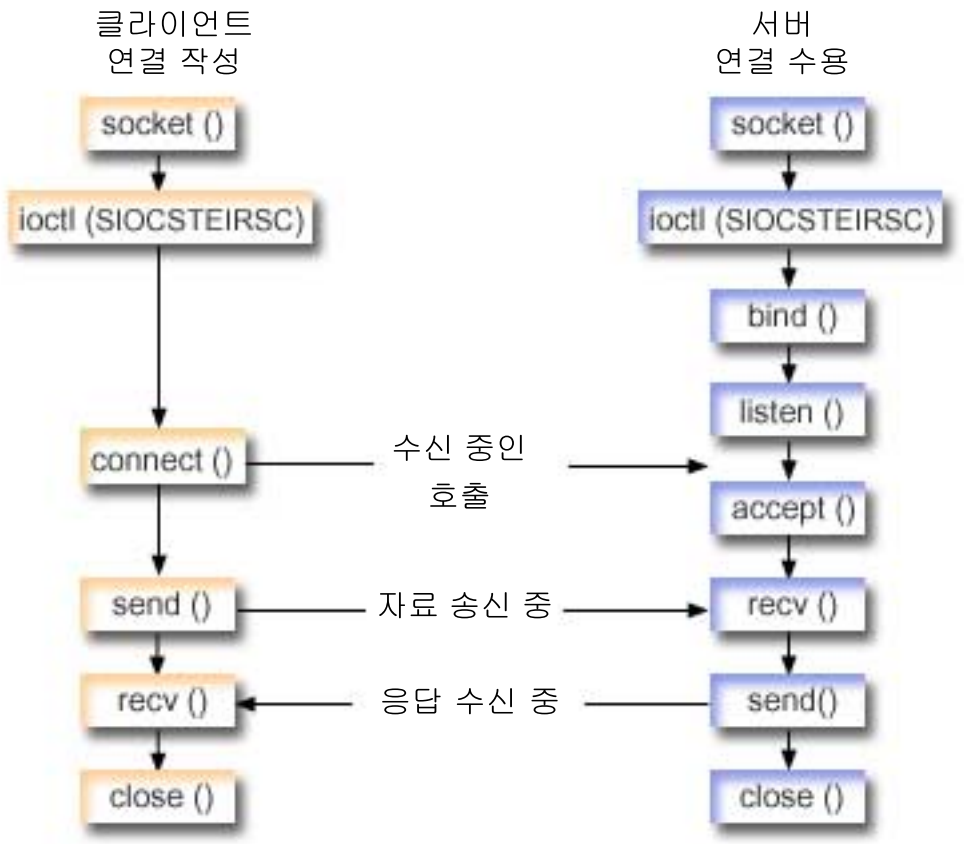
AF_TELEPHONY 소켓은 연결 지향(SOCK_STREAM 유형) 소켓 전용으로 지원됩니다. 이러한 소켓에는 다른 연결 지향 프로토콜의 것과 유사한 의미론 및 기능이 있습니다. 주요 차이점은 전화 통신 정의역의 연결이 기본 전화 연결보다 더 나은 신뢰성을 제공하지 못한다는 점입니다. 전달을 보증하려면 이 패밀리를 사용하는 팩스 어플리케이션과 같이 어플리케이션 레벨에서 이 전달을 완료해야 합니다. 또한, 대역폭을 벗어난 자료의 개념은 전화 통신 주소 패밀리에 지원되지 않습니다.

AF_TELEPHONY 소켓은 연결이 시작되거나 완료되기 전에 네트워크 전화 통신 장치(논리적으로 전화)와 연관되어야 합니다. 특수 **ioctl()** 명령 **SIOCSTELRSC**(전화 통신 자원 설정)를 사용하여 이를 연관시키십시오. 이러한 장치를 구성하여 이 명령을 발행하기 전에 사용할 준비가 되어 있어야 합니다.

SIOCSTELRSC **ioctl()**을 호출하기 전에, 어플리케이션에서는 장치명을 분석해야 합니다. 장치명은 시스템 포인터로 해석되어야 하며 SIOCSTELRSC 명령에 대한 입력으로 이 포인터를 사용해야 합니다.

장치는 소켓이 닫힐 때까지 소켓과 연관된 상태로 남아 있습니다. 결과적으로 하나의 소켓에 둘 이상의 장치를 연결할 수 있습니다. 이러한 복수 연관을 통해 어플리케이션은 단일 소켓을 통해 여러 장치에서의 호출을 청취하거나 응답할 수 있습니다.

다음 그림은 AF_TELEPHONY 주소 패밀리와 함께 사용되는 소켓 호출의 관계를 설명합니다. AF_UNIX 주소 패밀리를 사용하기 위해 환경을 설정하는 것에 대한 세부사항은 소켓 프로그래밍의 전제조건을 참조하십시오.



이벤트의 소켓 흐름: AF_TELEPHONY 주소 패밀리를 사용하는 클라이언트

예: AF_TELEPHONY 연결 작성에서는 다음 기능 호출 순서를 사용합니다.

1. **socket()** 함수는 종료점을 표시하는 소켓 설명자를 리턴합니다.
2. 시스템 포인터에 대해 장치명을 분석하고 요구에 대한 구조를 채워 소켓을 장치와 연관시켜야 합니다. **ioctl()** 함수를 발행하여 장치 및 시스템 포인터를 연관시키십시오.
3. 소켓 설명자가 수신된 후에 서버와의 연결 설정에 **connect()** 함수가 사용됩니다.
4. **send()** 함수는 클라이언트로 송신 버퍼의 내용을 송신합니다.
5. **recv()** 함수는 클라이언트 어플리케이션으로부터 자료를 수신합니다.
6. **close()** 함수는 열린 소켓 설명자를 닫습니다.

이벤트의 소켓 흐름: AF_TELEPHONY 주소 패밀리를 사용하는 서버 어플리케이션

예: AF_TELEPHONY 연결 허용에서는 다음 기능 호출 순서를 사용합니다.

1. **socket()** 함수는 종료점을 표시하는 소켓 설명자를 리턴합니다.
2. 시스템 포인터에 대해 장치명을 분석하고 요구에 대한 구조를 채워 소켓을 장치와 연관시켜야 합니다. **ioctl()** 함수를 발행하여 장치 및 시스템 포인터를 연관시키십시오.
3. 소켓 설명자가 수신된 후에 서버와의 연결 설정에 **connect()** 함수가 사용됩니다.
4. **listen()**은 서버가 수신 클라이언트 연결을 허용할 수 있도록 합니다.

| 5. 서버에서는 수신 연결 요구를 승인하기 위해 **accept()** 함수를 사용합니다.
| 6. **recv()** 함수는 클라이언트 어플리케이션으로부터 자료를 수신합니다.
| 7. **send()** 함수는 클라이언트로 송신 버퍼의 내용을 송신합니다.
| 8. **close()** 함수는 열린 소켓 설명자를 닫습니다.

예: AF_TELEPHONY 연결 작성

프로그램은 전화 정의역 소켓을 통해 서로 통신할 수 있습니다. 소켓이 클라이언트와의 연결을 만들 수 있게 하려면 다음 코드를 사용하십시오. 코드 예를 사용하는 것에 대한 정보는 코드 면책사항을 참조하십시오.

```
/*********************************************************************/
/* This sample program provides code to make AF_TELEPHONY connections.    */
/*********************************************************************/

#include  <stdio.h>                 /* String Functions */
#include  <string.h>                /* String Functions */
#include  <miptrnam.h>              /* Pointer types */

#include <sys/socket.h>             /* Sockets */
#include <nettel/tel.h>             /* Telephony address family */
#include <errno.h>                  /* Error codes */
#include <sys/ioctl.h>              /* Error codes */

int main() {
  /*****************************************************************/
  /* Miscellaneous declares                                        */
  /*****************************************************************/
  int xSock, xRC, xLength;

  /*****************************************************************/
  /* Resolve device name to system pointer                         */
  /*****************************************************************/
  _SYSPTR pDev;                     /* System pointer to device   */
  _RSLV_Template_T xTemp;           /* Template for resolve       */
  char pName[]="FRED                        "; /* Device name     */
  struct TelResource xResource;     /* SIOCSTELRSC structure      */

  /*****************************************************************/
  /* Socket address structure                                      */
  /*****************************************************************/
  struct sockaddr_tel xAddr;

  /*****************************************************************/
  /* Buffers                                                       */
  /*****************************************************************/
  char pSendBuffer[1024];
  char pRecvBuffer[1024];

  /*****************************************************************/
  /* Open a socket                                                 */
  /*****************************************************************/
 xSock = socket(AF_TELEPHONY,SOCK_STREAM,0);
 if (xSock<0) {
    perror("socket() failed");
    return(-1);
  }
```

```
  /*****************************************************************/
  /* Associate the socket with a device                            */
  /* ...resolve the device name to a system pointer                */
  /* ...fill in the structure for this request                     */
  /* ...issue the ioctl to perform the association                 */
  /*****************************************************************/
 memset(&xTemp,0x00,sizeof(xTemp));
 memcpy(xTemp.Obj.Name, pName, 30);
  xTemp.Obj.Type_Subtype = WLI_DEVD;
 xTemp.Auth = _AUTH_NONE;
 _RSLVSP2(&pDev,&xTemp);

 memset(&xResource,0x00,sizeof(xResource));
xResource.trCount=1;
  xResource.trResourceList=&pDev;


  xRC=ioctl(xSock,SIOCSTELRSC,&xResource);
  if (xRC<0) {
  perror("ioctl() failed");
    close(xSock);
    return(-1);
  }

  /*****************************************************************/
  /* Connect to a remote resource (dial a call)                    */
  /*****************************************************************/
  memset(&xAddr,0x00,sizeof(xAddr));
 xAddr.stel_family=AF_TELEPHONY;
xAddr.stel_addr.t_len=11;
memcpy(xAddr.stel_addr.t_addr,"18005551212",11);
  xRC=connect(xSock,(struct sockaddr*)&xAddr,sizeof(xAddr));
  if (xRC<0) {
    perror("connect() failed");
    close(xSock);
    return(-1);
  }

  /*****************************************************************/
  /* Send the contents of the send buffer                          */
  /*****************************************************************/
 xRC=send(xSock,pSendBuffer,1024,0);
  if (xRC<0) {
      perror("send() failed");
    close(xSock);
    return(-1);
  }

  /*****************************************************************/
  /* Receive a reply                                               */
  /*****************************************************************/
 xRC=recv(xSock,pRecvBuffer,1024,0);
  if (xRC<0) {
      perror("recv() failed");
    close(xSock);
    return(-1);
   }
```

```
  /*****************************************************************/
  /* All done, close and return                                    */
  /*****************************************************************/
    close(xSock);
  return(0);
}
```

예: AF_TELEPHONY 연결 승인

AF_TELEPHONY 주소 패밀리는 전화 번호를 사용하여 소켓을 식별하는 어플리케이션에 사용됩니다. 이 주소 패밀리는 주로 팩시밀리 어플리케이션에서 사용됩니다. 프로그램은 전화 정의역 소켓을 통해 서로 통신할 수 있습니다. 소켓이 서버로부터의 연결을 승인할 수 있게 하려면 다음 코드를 사용하십시오. 코드 예를 사용하는 것에 대한 정보는 코드 면책사항을 참조하십시오.

```
/*********************************************************************/
/* This sample program provides code to accept AF_TELEPHONY connections.  */
/*********************************************************************/
#include  <stdio.h>                  /* String Functions */
#include  <string.h>                 /* String Functions */
#include  <miptrnam.h>               /* Pointer types */

#include <sys/socket.h>              /* Sockets */
#include <nettel/tel.h>              /* Telephony address family */
#include <errno.h>                   /* Error codes */
#include <sys/ioctl.h>               /* Error codes */

int main() {

  /*****************************************************************/
  /* Micellaneous declares                                         */
  /*****************************************************************/
  int xSock,xNewSock,xRC,xLength;

  /*****************************************************************/
  /* Resolve device name to system pointer data areas              */
  /*****************************************************************/
  _SYSPTR pDev;                      /* System pointer to device   */
  _RSLV_Template_T xTemp;            /* Template for resolve       */
  char pName[]="GEORGE                       "; /* Device name    */
  struct TelResource xResource;      /* SIOCSTELRSC structure      */

  /*****************************************************************/
  /* Socket address structure                                      */
  /*****************************************************************/
  struct sockaddr_tel xAddr;

  /*****************************************************************/
  /* Buffers                                                       */
  /*****************************************************************/
  char pSendBuffer[1024];
  char pRecvBuffer[1024];

  /*****************************************************************/
  /* Open a socket                                                 */
  /*****************************************************************/
 xSock = socket(AF_TELEPHONY,SOCK_STREAM,0);
 if (xSock<0) {
```

```
    perror("socket() failed");
    return(-1);
  }

  /*****************************************************************/
  /* ...first, resolve the device name to a system pointer         */
  /*  ...next, fill in the structure for this request              */
  /* ...finally, issue the ioctl to perform the association        */
  /*****************************************************************/
 memset(&xTemp,0x00,sizeof(xTemp));
 memcpy(xTemp.Obj.Name, pName, 30);
  xTemp.Obj.Type_Subtype = WLI_DEVD;
 xTemp.Auth = _AUTH_NONE;
 _RSLVSP2(&pDev,&xTemp);

 memset(&xResource,0x00,sizeof(xResource));
xResource.trCount=1;
  xResource.trResourceList=&pDev;

  xRC=ioctl(xSock,SIOCSTELRSC,&xResource);
  if (xRC<0) {
  perror("ioctl() failed");
    close(xSock);
    return(-1);
  }

  /*****************************************************************/
  /* Bind to a local number (using TELADDR_ANY means to accept     */
  /* calls for any number in the inbound connection list's entries)*/
  /*****************************************************************/
  memset(&xAddr,0x00,sizeof(xAddr));
 xAddr.stel_family=AF_TELEPHONY;
 xAddr.stel_addr.t_len=TELADDR_LEN;
 memcpy(xAddr.stel_addr.t_addr,TELADDR_ANY,TELADDR_LEN);
  xRC=bind(xSock,(struct sockaddr*)&xAddr,sizeof(xAddr));
  if (xRC<0) {
    perror("bind() failed");
    close(xSock);
    return(-1);
  }

  /*****************************************************************/
  /* Listen for incoming calls                                     */
  /*****************************************************************/
 xRC=listen(xSock,5);
  if (xRC<0) {
    perror("listen() failed");
    close(xSock);
    return(-1);
  }
  /*****************************************************************/
  /* Accept an incoming call                                       */
  /*****************************************************************/
  memset(&xAddr,0x00,sizeof(xAddr));
  xLength = sizeof(xAddr);
  xNewSock=accept(xSock,(struct sockaddr*)&xAddr,&xLength);
 if (xNewSock<0) {
      perror("accept() failed");
```

```
    close(xSock);
    return(-1);
  }

  /*************************************************************/
  /* Receive some data                                         */
  /*************************************************************/
xRC=recv(xNewSock,pRecvBuffer,1024,0);
  if (xRC<0) {
      perror("recv() failed");
    close(xSock);
    close(xNewSock);
    return(-1);
  }

  /*************************************************************/
  /* Send a reply                                              */
  /*************************************************************/
xRC=send(xNewSock,pSendBuffer,1024,0);
  if (xRC<0) {
      perror("send() failed");
    close(xSock);
    close(xNewSock);
    return(-1);
  }

  /*************************************************************/
  /* All done, close both sockets and return                   */
  /*************************************************************/
    close(xSock);
    close(xNewSock);
  return(0);
}
```

제 7 장 소켓 개념

다음 주제에서는 소켓과 소켓 작동 방법에 대한 일반적인 논의의 범위를 넘는 확장 소켓 개념이 논의됩니다. 여기에서는 보다 크고 보다 복잡한 네트워크의 소켓 어플리케이션을 설계하는 방법을 제공합니다. 다음 개념 각각은 해당 샘플 프로그램에 링크되어 있습니다.

- 비동기 I/O
- 보안 소켓
- 클라이언트 SOCKS 지원
- 스레드 안전성
- 비블록화 I/O
- 신호
- IP 멀티캐스팅
- 파일 자료 전송--send_file() 및 accept_and_recv()
- 대역폭을 벗어난 자료
- I/O 멀티플렉싱--select()
- 소켓 네트워크 함수
- 정의역명 시스템(DNS) 지원
- BSD 호환성
- 프로세스간의 설명자 전달--sendmsg() 및 recvmsg()

비동기 I/O

비동기 I/O API에서는 높은 동시 및 메모리 효율적인 I/O를 수행하기 위해 스레드 클라이언트 서버 모델에 메소드를 제공합니다. 이전 스레드 클라이언트/서버 모델에서 일반적으로 두 개의 I/O 모델이 보급되었습니다. 첫 번째 모델은 클라이언트 연결당 하나의 스레드가 전용됩니다. 첫 번째 모델은 너무 많은 스레드를 소비하며 실질적인 무활동 및 활동 비용을 초래할 수 있습니다. 두 번째 모델은 큰 클라이언트 연결 세트에서 **select()** API를 발행하고 준비가 되어 있는 클라이언트 연결 및 요구를 스레드에 위임하여 스레드 수를 최소화합니다. 두 번째 모델에서는 후속적인 여분의 작업 양의 원인이 될 수 있는 각 후속 선택시 선택하거나 표시해야 합니다.

비동기 I/O 및 중첩 I/O는 제어가 사용자 어플리케이션으로 리턴된 후에 사용자 버퍼 사이에서 자료를 전달하여 이러한 문제점을 모두 분석합니다. 비동기 I/O는 자료를 읽을 수 있거나 연결이 자료를 전송할 준비가 되었을 때 이러한 작업자 스레드를 작업자에게 통지합니다.

비동기 I/O 장점

- 시스템 자원을 보다 효율적으로 사용합니다.
  사용자 버퍼 사이의 자료 사본은 요구를 시작하는 어플리케이션에 대해 비동기입니다. 이러한 중첩된 처리는 복수 프로세서를 효율적으로 사용할 수 있게 만들며 대부분의 경우 시스템 버퍼가 자료 도착시 재사용을 위해 해제되므로 페이징 비율을 향상시킵니다.
- 프로세스/스레드 대기 시간을 최소화합니다.
- 클라이언트 요구에 즉시 서비스를 제공합니다.
- 평균하여 무활동 및 활동 비용을 배웁니다.
- "파손된 어플리케이션"을 효율적으로 처리합니다.
- 더 나은 스케일러빌리티를 제공합니다.
- 대량 자료 전송 처리의 가장 효율적인 메소드를 제공합니다.
  QsoStartRecv() API의 fillBuffer 플래그는 오퍼레이팅 시스템에 비동기 I/O를 완료하기 전에 대량의 자료를 확보하도록 알립니다. 대량의 자료는 하나의 비동기 조작으로 송신될 수도 있습니다.
- 필요한 스레드 수를 최소화합니다.
- 선택적으로 타이머를 사용하여 비동기로 이 조작을 완료하는 데 허용되는 최대 시간을 지정할 수 있습니다. 서버는 일정 시간 동안 유휴 상태일 경우 클라이언트 연결을 닫습니다. 비동기 타이머를 사용하면 서버가 이 시간 제한을 강제할 수 있습니다.
- **gsk_secure_soc_startInit()** API를 사용하여 비동기로 보안 세션을 초기화합니다.

표 13. 비동기 I/O API

함수	설명
gsk_secure_soc_startInit()	SSL 환경 및 보안 세션에 속성 세트를 사용하여 비동기로 보안 세션 협상을 시작합니다. **주:** 이 API만 AF_INET 또는 AF_INET6 주소 패밀리와 SOCK_STREAM 유형의 소켓을 지원합니다.
gsk_secure_soc_startRecv()	보안 세션에서 비동기 수신 조작을 시작합니다. **주:** 이 API만 AF_INET 또는 AF_INET6 주소 패밀리와 SOCK_STREAM 유형의 소켓을 지원합니다.
gsk_secure_soc_startSend()	보안 세션에서 비동기 송신 조작을 시작합니다. **주:** 이 API만 AF_INET 또는 AF_INET6 주소 패밀리와 SOCK_STREAM 유형의 소켓을 지원합니다.
QsoCreateIOCompletionPort()	완료된 비동기 중첩 I/O 조작의 공통 대기점을 작성합니다. **QsoCreateIOCompletionPort()** 함수는 대기점을 나타내는 포트 핸들을 리턴합니다. 이 핸들은 **QsoStartRecv()**, **QsoStartSend()**, **QsoStartAccept()**, **gsk_secure_soc_startRecv()** 또는 **gsk_secure_soc_startSend()** 함수에 지정되어 비동기 중첩 I/O 조작을 시작합니다. 또한, 이 핸들은 **QsoPostIOCompletion()**에 사용되어 연관된 I/O 완료 포트에서 이벤트를 전송합니다.
QsoDestroyIOCompletionPort()	I/O 완료 포트를 훼손시킵니다.
QsoWaitForIOCompletionPort()	완료된 중첩 I/O 조작을 기다립니다. I/O 완료 포트는 이 대기점을 나타냅니다.

표 13. 비동기 I/O API (계속)

QsoStartAccept()	비동기 승인 조작을 시작합니다. **주:** 이 API만 AF_INET 또는 AF_INET6 주소 패밀리와 SOCK_STREAM 유형의 소켓을 지원합니다.
QsoStartRecv()	비동기 수신 조작을 시작합니다. **주:** 이 API만 AF_INET 또는 AF_INET6 주소 패밀리와 SOCK_STREAM 유형의 소켓을 지원합니다.
QsoStartSend()	비동기 송신 조작을 시작합니다. **주:** 이 API만 AF_INET 또는 AF_INET6 주소 패밀리와 SOCK_STREAM 소켓 유형의 소켓을 지원합니다.
QsoPostIOCompletion()	어플리케이션에서 일부 함수 또는 활동이 발생했다는 것을 완료 포트에 알릴 수 있습니다.

비동기 I/O 작동 방법

어플리케이션에서는 **QsoCreateIOCompletionPort()** API를 사용하여 I/O 완료 포트를 작성합니다. 이 API는 비동기 I/O 요구 완료를 스케줄하고 기다리는 데 사용될 수 있는 핸들을 리턴합니다. 어플리케이션은 I/O 완료 포트 핸들을 지정하여 입력 또는 출력 함수를 시작합니다. I/O가 완료되면 상태 정보 및 어플리케이션 정의 핸들이 지정된 I/O 완료 포트로 전송됩니다. I/O 완료 포트에 대한 전송은 대기 중인 가능한 다수의 스레드 중 하나를 정확하게 활동시킵니다. 어플리케이션에서는 다음을 수신합니다.

- 원래 요구에 제공된 버퍼
- 해당 버퍼 사이에서 처리된 자료 길이
- I/O 조작이 완료된 유형 표시
- 초기 I/O 요구에서 전달된 어플리케이션 정의 핸들

이 어플리케이션 핸들은 단순히 클라이언트 연결을 식별하는 소켓 설명자나 클라이언트 연결 상태에 대한 확장 정보를 포함하는 기억장치에 대한 포인터가 될 수 있습니다. 조작이 완료되었고 어플리케이션 핸들이 전달되었으므로, 작업자 스레드는 클라이언트 연결을 완료할 다음 단계를 판별합니다. 이러한 완료된 비동기 조작을 처리하는 작업자 스레드는 다수의 여러 클라이언트 요구를 처리할 수 있으며 하나만 결합되지 않습니다. 사용자 버퍼 사이의 복사가 서버 프로세스에 비동기적으로 발생하므로 클라이언트 요구의 대기 시간이 줄어듭니다. 이는 복수 프로세서가 있는 시스템에서 유익합니다.

비동기 I/O를 사용하는 단순 서버 모델의 예는 예: 비동기 I/O 사용을 참조하십시오.

보안 소켓

현재 OS/400은 iSeries에서 보안 소켓 어플리케이션을 작성하는 두 가지 메소드를 지원합니다. SSL_ API 및 글로벌 보안 툴킷(GSKit) API는 열린 통신 네트워크(대부분의 경우 인터넷)를 통해 통신 보호를 제공합니다. API는 클라이언트/서버 어플리케이션이 도청, 간섭 및 메세지 위조 등을 방지하기 위해 설계된 방식으로 통신할 수 있게 합니다. 서버 및 클라이언트 인증을 모두 지원하며 모두 어플리케이션이 보안 소켓 층(SSL) 프로토콜을 사용할 수 있도록 합니다. 그러나 GSKit API가 모든 IBM @server 플랫폼에 걸쳐 지원되는 반면에 SSL_ API는 OS/400 오퍼레이팅 시스템에 고유합니다. 플랫폼간 상호운영성을 보장하기 위해 보안 소켓 연결을 위해 어플리케이션을 개발할 때 GSKit API를 사용할 것을 권장합니다.

이런 API 각각에 대한 설명은 다음 주제를 참조하십시오.

- 글로벌 보안 툴킷(GSKit) API
- SSL_ APIs

보안 소켓 API 오류 코드 메세지 주제에서는 보안 소켓 API에 발생할 수 있는 일반적인 오류 코드 메세지 리스트를 제공합니다.

보안 소켓 개요

Netscape가 최초로 개발한 보안 소켓 층(SSL) 프로토콜은 어플리케이션에 보안 통신을 제공하기 위해 전송 제어 프로토콜(TCP)과 같은 신뢰성 있는 전송의 맨 위에서 사용될 목적의 계층 프로토콜입니다. 보안 통신을 필요로 하는 다수의 어플리케이션 중 일부는 HTTP, FTP, SMTP 및 TELNET입니다.

SSL에서 작동 가능한 어플리케이션은 보통 SSL에서 작동 가능하지 않은 어플리케이션과 다른 포트를 사용해야 합니다. 예를 들면 SSL 작동 브라우저는 ″HTTP″가 아니라 ″HTTPs″로 시작하는 URL(Universal Resource Locator)을 사용하여 SSL 작동 HTTP(하이퍼텍스트 전송 프로토콜) 서버에 액세스합니다. 대부분의 경우 ″HTTPs″의 URL은 표준 HTTP 서버에서 사용하는 포트 80 대신에 서버 시스템의 포트 443에 대한 연결을 열고자 합니다.

여러 개의 SSL 프로토콜 버전이 정의되어 있습니다. 최신 버전인 전송층 보안(TLS) 버전 1.0에서는 SSL 버전 3.0에서 향상되는 업그레이드를 제공합니다. iSeries 고유의 SSL_ API와 GSKit API 모두 TLS 버전 1.0, SSL 버전 3.0과 호환성이 있는 TLS 버전 1.0, SSL 버전 3.0, SSL 버전 2.0 및 2.0과 호환성이 있는 SSL 버전 3.0을 모두 지원합니다. TLS 버전 1.0에 대한 자세한 내용은 인터넷 엔지니어링 임시 전담 팀(IETF) RFC 2246, ″Transport Layer Security″ 을 참조하십시오.

글로벌 보안 툴킷(GSKit) API

글로벌 보안 툴킷(GSKit)은 어플리케이션에서 SSL 작동될 수 있는 프로그램 가능 인터페이스의 세트입니다. SSL_ API와 같이, GSKit API를 사용하여 소켓 어플리케이션 프로그램에서 SSL 및 TLS 함수에 액세스할 수 있습니다. 그러나 GSKit API가 모든 IBM @server 플랫폼에 걸쳐 지원되며 이전 SSL_ API에서 보다 프로그래밍하기가 수월합니다. 또한 보안 소켓 세션의 비동기 인스턴스를 작성하기 위해 새로운 GSKit API가 추가되었습니다. 이 API는 수신 요구 수가 너무 많고 다중 작업이 필요한 경우 다중 클라이언트를 처리할 수 있는 보안 연결을 제공합니다. 그러나 이 API는 OS/400에만 고유하며 다른 @server 플랫폼에 이식할 수 없습니다.

주: 이러한 API만 AF_INET 또는 AF_INET6 주소 패밀리와 SOCK_STREAM 유형의 소켓을 지원합니다.

다음 표에는 GSKit API가 설명되어 있습니다.

표 14. 글로벌 보안 툴킷 API

함수	설명
gsk_attribute_get_buffer()	인증 저장 파일, 인증 저장 암호, 어플리케이션 ID 및 암호와 같은 보안 세션이나 SSL 환경에 대한 특정 문자 스트링 정보를 확보합니다.

표 14. 글로벌 보안 툴킷 API (계속)

gsk_attribute_get_cert_info()	보안 세션이나 SSL 환경에 대한 서버 또는 클라이언트 인증에 대한 특정 정보를 확보합니다.
gsk_attribute_get_enum_value()	보안 세션이나 SSL 환경에 대한 특정의 열거된 자료값을 확보합니다.
gsk_attribute_get_numeric_value()	보안 세션이나 SSL 환경에 대한 특정 숫자 정보를 확보합니다.
gsk_attribute_set_buffer()	지정된 보안 세션이나 SSL 환경 내부의 값으로 지정된 버퍼 속성을 설정합니다.
gsk_attribute_set_enum()	지정된 열거 유형 속성을 보안 세션이나 SSL 환경에서 열거된 값으로 설정합니다.
gsk_attribute_set_numeric_value()	보안 세션이나 SSL 환경의 특정 숫자 정보를 설정합니다.
gsk_environment_close()	SSL 환경을 닫고 환경과 연관된 모든 기억장치를 해제합니다.
gsk_environment_init()	필요한 속성을 설정한 후에 SSL 환경을 초기화합니다.
gsk_environment_open()	후속적인 gsk 호출에 저장되고 사용되어야 하는 SSL 환경 핸들을 리턴합니다.
gsk_secure_soc_close()	보안 세션을 닫은 후 해당 보안 세션의 모든 연관 자원을 해제합니다.
gsk_secure_soc_init()	SSL 환경 및 보안 세션이 속성 세트를 사용하여 보안 세션을 조정합니다.
gsk_secure_soc_misc()	보안 세션의 기타 함수를 수행합니다.
gsk_secure_soc_open()	보안 세션의 기억장치를 확보하고 속성의 디폴트 값을 설정하며 보안 세션 관련 함수 호출에 저장되어 사용되어야 하는 핸들을 리턴합니다.
gsk_secure_soc_read()	보안 세션에서 자료를 수신합니다.
gsk_secure_soc_startInit()	SSL 환경 및 보안 세션에 속성 세트를 사용하여 비동기로 보안 세션 협상을 시작합니다.
gsk_secure_soc_write()	보안 세션의 자료를 기록합니다.
gsk_secure_soc_startRecv()	보안 세션에서 비동기 수신 조작을 시작합니다.
gsk_secure_soc_startSend()	보안 세션에서 비동기 송신 조작을 시작합니다.
gsk_strerror()	오류 메세지와 GSK API 호출에서 리턴되는 리턴 값을 설명하는 연관 텍스트 스트링을 검색합니다.

소켓과 GSKit API를 사용하는 어플리케이션에는 다음 요소가 포함됩니다.

1. 소켓 설명자를 확보하기 위한 **socket()**에 대한 호출
2. SSL 환경에 대한 핸들을 확보하기 위한 **gsk_environment_open()** 호출
3. SSL 환경의 속성을 설정하기 위한 하나 이상의 gsk_attribute_set_xxxxx() 호출. 최소한 **gsk_attribute_set_buffer()**를 호출하여 GSK_OS400_APPLICATION_ID 값이나 GSK_KEYRING_FILE 값을 설정하십시오이들 중 하나만 설정되어야 합니다. GSK_OS400_APPLICATION_ID 값을 사용하는 것이 더 좋습니다. 또한, **gsk_attribute_set_enum()**을 사용하여 어플리케이션 유형(클라이언트 또는 서버), GSK_SESSION_TYPE을 설정해야 합니다.
4. SSL 처리의 이 환경을 초기화하고 이 환경을 사용하여 실행할 모든 SSL 세션의 SSL 보안 정보를 설정하기 위한 **gsk_environment_init()** 호출.

5. 연결을 활성화하기 위한 소켓 호출. **connect()** 를 호출하여 클라이언트 프로그램의 연결을 활성화하거나 **bind()**, **listen()** 및 **accept()** 를 호출하여 서버가 수신 연결 요구를 승인할 수 있게 합니다
6. 보안 세션에 대한 핸들을 확보하기 위한 **gsk_secure_soc_open()** 호출
7. 보안 세션의 속성을 설정하기 위한 하나 이상의 gsk_attribute_set_xxxxx() 호출. 최소한, 이 보안 세션과 특정 소켓을 연관시키기 위한 **gsk_attribute_set_numeric_value()** 호출
8. 암호 매개변수의 SSL 핸드쉐이크 조정을 시작하기 위한 **gsk_secure_soc_init()** 호출.

   주: 일반적으로 서버 프로그램에서는 SSL 핸드쉐이크 완료에 대한 인증을 제공해야 합니다. 또한, 서버는 서버 인증과 연관된 개인 키와 인증이 저장된 키 데이터베이스 파일에 액세스할 수도 있습니다. 일부 경우 클라이언트에서는 SSL 핸드쉐이크 처리시 인증도 제공해야 합니다. 이는 클라이언트가 연결 중인 서버에 클라이언트 인증이 작동 가능한 경우에 발생합니다. **gsk_attribute_set_buffer** (GSK_OS400_APPLICATION_ID) 또는 **gsk_attribute_set_buffer** (GSK_KEYRING_FILE) API 호출은 핸드쉐이크시 사용되는 인증 및 개인 키를 확보하는 키 데이터베이스 파일을 식별합니다(유사하지 않은 방법인 경우에도).
9. **gsk_secure_soc_read()** 및 **gsk_secure_soc_write()**를 호출하여 자료를 송수신합니다
10. 보안 세션을 종료하기 위한 **gsk_secure_soc_close()** 호출
11. SSL 환경을 닫기 위한 **gsk_environment_close()** 호출
12. 연결된 소켓을 무효로 하기 위한 **close()** 호출.

GSKit API를 사용하는 샘플 프로그램은 다음 예를 참조하십시오.

- 예: 비동기 자료 수신을 사용하여 보안 서버 설정
- 예: 비동기 핸드쉐이크를 사용하는 보안 서버 설정
- 예: 글로벌 보안 툴킷(GSKit) API를 사용하여 보안 클라이언트 설정

SSL_ API

SSL_ API는 프로그래머가 iSeries에서 보안 소켓 어플리케이션을 작성할 수 있도록 합니다. GSKitAPI와 달리 SSL_ API는 OS/400 시스템에만 고유합니다. 다음 표에서는 OS/400 구현에서 지원되는 9개의 SSL_ API를 설명합니다. Information Center의 API 정보에 나열된 개별 API에 대한 세부사항을 알려면 링크를 사용하십시오.

표 15. SSL_ API

함수	설명
SSL_Create()	지정된 소켓 설명자에 대해 SSL 지원이 작동 가능합니다.
SSL_Destroy()	지정된 SSL 세션 및 소켓에 대해 SSL 지원을 종료합니다.
SSL_Handshake()	SSL 핸드쉐이크 프로토콜을 시작합니다.
SSL_Init()	SSL의 현재 작업을 초기화하고 현재 작업의 SSL 보안 정보를 설정합니다. **주:** **SSL_Init()** 또는 **SSL_Init_Application()** API는 SSL이 사용되기 전에 프로세스에서 실행되어야 합니다.

표 15. SSL_ API (계속)

SSL_Init_Application()	SSL의 현재 작업을 초기화하고 현재 작업의 SSL 보안 정보를 설정합니다. **주: SSL_Init()** 또는 **SSL_Init_Application()** API는 SSL이 사용되기 전에 프로세스에서 실행되어야 합니다.
SSL_Read()	SSL 작동 소켓 설명자에서 자료를 수신합니다.
SSL_Write()	SSl 작동 소켓 설명자에 자료를 기록합니다.
SSL_Strerror()	SSL 실행시 오류 메세지를 검색합니다.
SSL_Perror()	SSL 오류 메세지를 인쇄합니다.

소켓과 SSL_ API를 사용하는 어플리케이션에는 다음과 같은 요소가 포함됩니다.

- 소켓 설명자를 확보하기 위한 **socket()**에 대한 호출
- SSL 처리의 작업 환경을 초기화하고 현재 작업에서 실행될 모든 SSL 세션의 SSL 보안 정보를 설정하기 위한 **SSL_Init()** 또는 **SSL_Init_Application()** 호출. 이러한 API 중 하나만이 사용되어야 합니다. **SSL_Init_Application()** API를 사용하는 것이 더 좋습니다.
- 연결을 활성화하기 위한 소켓 호출. **connect()**를 호출하여 클라이언트 프로그램의 연결을 활성화하거나 **bind()**, **listen()** 및 **accept()**를 호출하여 서버가 수신 연결 요구를 승인할 수 있게 합니다.
- 연결된 소켓에 대한 SSL 지원을 사용하기 위한 **SSL_Create()**에 대한 호출
- 암호화 매개변수의 SSL 핸드쉐이크 조정을 초기설정하기 위한 **SSL_Handshake()**에 대한 호출

  주: 일반적으로 서버 프로그램에서는 SSL 핸드쉐이크 완료에 대한 인증을 제공해야 합니다. 또한, 서버는 서버 인증과 연관된 개인 키와 인증이 저장된 키 데이터베이스 파일에 액세스할 수도 있습니다. 일부 경우 클라이언트에서는 SSL 핸드쉐이크 처리시 인증도 제공해야 합니다. 이는 클라이언트가 연결 중인 서버에 클라이언트 인증이 작동 가능한 경우에 발생합니다. **SSL_Init()** 또는 **SSL_Init_Application()** API는 핸드쉐이크시 사용되는 인증 및 개인 키를 확보하는 키 데이터베이스 파일을 식별합니다(유사하지 않은 방법인 경우에도).
- 자료를 송수신하기 위한 **SSL_Read()** 및 **SSL_Write()**에 대한 호출
- 소켓에 대한 SSL 지원 작동을 불가능하게 하기 위한 **SSL_Destroy()**에 대한 호출
- 연결된 소켓을 소멸시키기 위한 **close()**에 대한 호출

이러한 SSL_ API를 사용하는 샘플 프로그램은 다음 샘플 프로그램을 참조하십시오.

- 예: SSL_ API를 사용하여 보안 서버 설정
- 예: SSL_ API를 사용하여 보안 클라이언트 설정

보안 소켓 API 오류 코드 메세지

다음 보안 소켓 오류 코드 메세지에 대한 정보에 액세스하려면 다음을 완료하십시오.

1. 명령행에 다음을 입력하십시오.

   ```
   DSPMSGD RANGE(XXXXXXX)
   ```

   여기서 XXXXXXX는 리턴 코드의 메세지 ID입니다. 예를 들어, 리턴 코드가 3인 경우 다음을 입력하십시오.

```
DSPMSGD RANGE(CPDBCB9)
```

2. 메세지 텍스트를 표시하려면 **1**을 선택하십시오.

표 16. 보안 소켓 API 오류 코드 메세지

리턴 코드	메세지 ID	상수 이름
0	CPCBC80	GSK_OK
4	CPCBC80	GSK_INSUFFICIENT_STORAGE
502	CPE3406	GSK_WOULD_BLOCK
1	CPDBCA1	GSK_INVALID_HANDLE
2	CPDBCB3	GSK_API_NOT_AVAILABLE
3	CPDBCB9	GSK_INTERNAL_ERROR
5	CPDBC95	GSK_INVALID_STATE
107	CPDBC98	GSK_KEYFILE_CERT_EXPIRED
201	CPDBCA4	GSK_NO_KEYFILE_PASSWORD
202	CPDBCB5	GSK_KEYRING_OPEN_ERROR
301	CPDBCA5	GSK_CLOSE_FAILED
402	CPDBC81	GSK_ERROR_NO_CIPHERS
403	CPDBC82	GSK_ERROR_NO_CERTIFICATE
404	CPDBC84	GSK_ERROR_BAD_CERTIFICATE
405	CPDBC86	GSK_ERROR_UNSUPPORTED_CERTIFICATE_TYPE
406	CPDBC8A	GSK_ERROR_IO
407	CPDBCA3	GSK_ERROR_BAD_KEYFILE_LABEL
408	CPDBCA7	GSK_ERROR_BAD_KEYFILE_PASSWORD
409	CPDBC9A	GSK_ERROR_BAD_KEY_LEN_FOR_EXPORT
410	CPDBC8B	GSK_ERROR_BAD_MESSAGE
411	CPDBC8C	GSK_ERROR_BAD_MAC
412	CPDBC8D	GSK_ERROR_UNSUPPORTED
414	CPDBC84	GSK_ERROR_BAD_CERT
415	CPDBC8B	GSK_ERROR_BAD_PEER
417	CPDBC92	GSK_ERROR_SELF_SIGNED
420	CPDBC96	GSK_ERROR_SOCKET_CLOSED
421	CPDBCB7	GSK_ERROR_BAD_V2_CIPHER
422	CPDBCB7	GSK_ERROR_BAD_V3_CIPHER
428	CPDBC82	GSK_ERROR_NO_PRIVATE_KEY
501	CPDBCA8	GSK_INVALID_BUFFER_SIZE
601	CPDBCAC	GSK_ERROR_NOT_SSLV3
602	CPDBCA9	GSK_MISC_INVALID_ID
701	CPDBCA9	GSK_ATTRIBUTE_INVALID_ID
702	CPDBCA6	GSK_ATTRIBUTE_INVALID_LENGTH
703	CPDBCAA	GSK_ATTRIBUTE_INVALID_ENUMERATION
705	CPDBCAB	GSK_ATTRIBUTE_INVALID_NUMERIC
6000	CPDBC97	GSK_OS400_ERROR_NOT_TRUSTED_ROOT

표 16. 보안 소켓 API 오류 코드 메세지 (계속)

6001	CPDBCB1	GSK_OS400_ERROR_PASSWORD_EXPIRED
6002	CPDBCC9	GSK_OS400_ERROR_NOT_REGISTERED
6003	CPDBCAD	GSK_OS400_ERROR_NO_ACCESS
6004	CPDBCB8	GSK_OS400_ERROR_CLOSED
6005	CPDBCCB	GSK_OS400_ERROR_NO_CERTIFICATE_AUTHORITIES
6007	CPDBCB4	GSK_OS400_ERROR_NO_INITIALIZE
6008	CPDBCAE	GSK_OS400_ERROR_ALREADY_SECURE
6009	CPDBCAF	GSK_OS400_ERROR_NOT_TCP
6010	CPDBC9C	GSK_OS400_ERROR_INVALID_POINTER
6011	CPDBC9B	GSK_OS400_ERROR_TIMED_OUT
6012	CPCBCBA	GSK_OS400_ASYNCHRONOUS_RECV
6013	CPCBCBB	GSK_OS400_ASYNCHRONOUS_SEND
6014	CPDBCBC	GSK_OS400_ERROR_INVALID_OVERLAPPEDIO_T
6015	CPDBCBD	GSK_OS400_ERROR_INVALID_IOCOMPLETIONPORT
6016	CPDBCBE	GSK_OS400_ERROR_BAD_SOCKET_DESCRIPTOR
6017	CPDBCBF	GSK_OS400_ERROR_CERTIFICATE_REVOKED
6018	CPDBC87	GSK_OS400_ERROR_CRL_INVALID
6019	CPCBC88	GSK_OS400_ASYNCHRONOUS_SOC_INIT
0	CPCBC80	Successful return
-1	CPDBC81	SSL_ERROR_NO_CIPHERS
-2	CPDBC82	SSL_ERROR_NO_CERTIFICATE
-4	CPDBC84	SSL_ERROR_BAD_CERTIFICATE
-6	CPDBC86	SSL_ERROR_UNSUPPORTED_CERTIFICATE_TYPE
-10	CPDBC8A	SSL_ERROR_IO
-11	CPDBC8B	SSL_ERROR_BAD_MESSAGE
-12	CPDBC8C	SSL_ERROR_BAD_MAC
-13	CPDBC8D	SSL_ERROR_UNSUPPORTED
-15	CPDBC84	SSL_ERROR_BAD_CERT (map to -4)
-16	CPDBC8B	SSL_ERROR_BAD_PEER (map to -11)
-18	CPDBC92	SSL_ERROR_SELF_SIGNED
-21	CPDBC95	SSL_ERROR_BAD_STATE
-22	CPDBC96	SSL_ERROR_SOCKET_CLOSED
-23	CPDBC97	SSL_ERROR_NOT_TRUSTED_ROOT
-24	CPDBC98	SSL_ERROR_CERT_EXPIRED
-26	CPDBC9A	SSL_ERROR_BAD_KEY_LEN_FOR_EXPORT
-91	CPDBCB1	SSL_ERROR_KEYPASSWORD_EXPIRED
-92	CPDBCB2	SSL_ERROR_CERTIFICATE_REJECTED
-93	CPDBCB3	SSL_ERROR_SSL_NOT_AVAILABLE
-94	CPDBCB4	SSL_ERROR_NO_INIT
-95	CPDBCB5	SSL_ERROR_NO_KEYRING
-97	CPDBCB7	SSL_ERROR_BAD_CIPHER_SUITE

표 16. 보안 소켓 API 오류 코드 메세지 (계속)

-98	CPDBCB8	SSL_ERROR_CLOSED
-99	CPDBCB9	SSL_ERROR_UNKNOWN
-1009	CPDBCC9	SSL_ERROR_NOT_REGISTERED
-1011	CPDBCCB	SSL_ERROR_NO_CERTIFICATE_AUTHORITIES
-9998	CPDBCD8	SSL_ERROR_NO_REUSE

클라이언트 SOCKS 지원

iSeries는 SOCKS 버전을 사용하여 SOCK_STREAM 소켓 유형의 AF_INET 주소 패밀리를 사용하는 프로그램이 방화벽 외부의 시스템에서 실행되는 서버 프로그램과 통신할 수 있도록 합니다. 방화벽은 네트워크 관리자가 보안 내부 네트워크와 비보안 외부 네트워크 사이에 위치시키는 보안이 매우 우수한 호스트입니다. 일반적으로 그러한 네트워크 구성은 보안 호스트에서 시작하는 통신이 비보안 네트워크로 라우트되지 않게 하며 이 반대의 경우에도 마찬가지입니다. 방화벽에 존재하는 프록시 서버는 보안 호스트와 비보안 네트워크 사이의 필요한 액세스를 관리합니다.

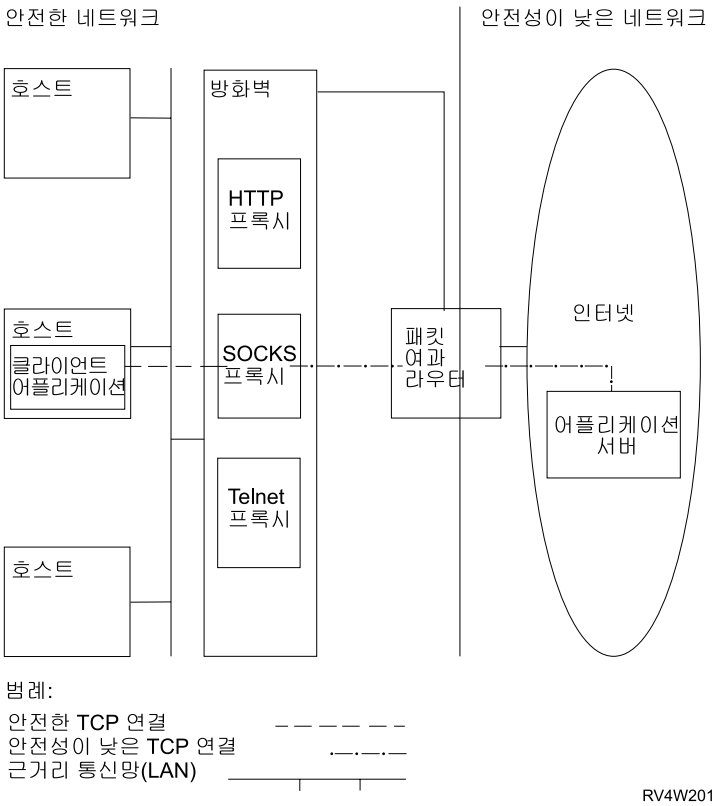
보안 내부 네트워크의 호스트상에서 실행되는 어플리케이션은 방화벽을 탐색하기 위한 요청을 방화벽 프록시 서버로 송신해야 합니다. 그러면 프록시 서버는 이러한 요구를 비보안 네트워크의 실제 서버로 전송하고 그 응답을 다시 시작 호스트의 어플리케이션으로 다시 릴레이할 수 있습니다. 프록시 서버의 일반적인 예로는 HTTP 프록시 서버를 들 수 있습니다. 프록시 서버는 HTTP 클라이언트에 대해 많은 타스크를 수행합니다.

- 외부 시스템에서 내부 네트워크를 숨깁니다.
- 외부 시스템에 의해 호스트가 직접 액세스되지 못하도록 합니다.
- 적절하게 설계되어 구성된 경우 외부에서 들어오는 자료를 필터링할 수 있습니다.

HTTP 프록시 서버는 HTTP 클라이언트만 처리합니다.

방화벽상에서 여러 프록시 서버를 실행하는 대신 사용되는 흔한 방법은 SOCKS 서버라는 더욱 강력한 프록시 서버를 실행하는 것입니다. SOCKS 서버는 소켓 API를 사용하여 설정된 TCP 클라이언트 연결에 대한 프록시의 역할을 할 수 있습니다. iSeries 클라이언트 SOCKS 지원의 주된 장점은 클라이언트 어플리케이션이 클라이언트 코드를 변경하지 않고 투명하게 SOCKS 서버에 액세스할 수 있다는 점입니다.

다음 그림은 방화벽에서 HTTP 프록시, telnet 프록시 및 SOCKS 프록시를 사용하는 일반적인 방화벽 배열을 보여줍니다. 인터넷상에서 서버에 액세스 중인 보안 클라이언트에 대해 두 개의 별도의 TCP 연결이 사용되었다는 점에 유의하십시오. 한 연결은 보안 호스트에서 SOCKS 서버로 향하고 또 다른 연결은 비보안 네트워크에서 SOCKS 서버로 향합니다.



SOCKS 서버를 사용하려면 보안 클라이언트 호스트에서 두 가지 조치가 필수입니다.

1. SOCKS 서버의 구성. 2000년 2월 15일, IBM에서는 SOCKS 서버 지원을 제공하는 iSeries용 Firewall 제품(5769-FW1)이 현재 V4R3 기능 이상 향상되지 않았다고 발표했습니다.
2. 보안 클라이언트 시스템에서 클라이언트 시스템에 SOCK 서버로 지정될 모든 아웃바운드 클라이언트 TCP 연결을 정의하십시오. iSeries Access 95 또는 Windows NT®의 iSeries Navigator 기능 아래에 있는 SOCKS 탭을 사용하여 보안 클라이언트 SOCKS 구성 항목을 정의할 수 있습니다. SOCKS 탭에는 클라이언트 SOCKS 지원에 대한 보안 클라이언트 시스템 구성에 대한 실제적인 도움말이 있습니다.

   클라이언트 SOCKS 지원을 구성하려면 다음을 수행하십시오.

   a. iSeries Navigator에서 **iSeries 서버 --> 네트워크 --> TCP/IP 구성**을 펼치십시오.
   b. **TCP/IP 구성**을 마우스 오른쪽 버튼으로 클릭하십시오.
   c. **등록 정보**를 클릭하십시오.
   d. **SOCKS** 탭을 클릭하십시오.
   e. SOCKS 페이지에 사용자의 연결 정보를 입력하십시오.

   주: 보안 클라이언트 SOCKS 구성 자료는 보안 클라이언트 호스트 시스템의 QUSRSYS 라이브러리에 있는 QASOSCFG 파일에 저장됩니다.

일단 구성되면 시스템에서는 SOCKS 페이지에 지정된 SOCKS 서버로 특정 아웃바운드 연결을 자동으로 지정합니다. 보안 클라이언트 어플리케이션을 변경할 필요가 없습니다. 요구를 수신하면 SOCKS 서버는 비보안 네트워크에 있는 서버로 별도의 외부 TCP/IP 연결을 설정합니다. 그러면 SOCKS 서버는 내부 및 외부 TCP/IP 연결 사이에 자료를 전달합니다.

주: 비보안 네트워크상의 리모트 호스트는 SOCKS 서버로 직접 연결합니다. 보안 클라이언트에 대한 직접 액세스 권한이 없습니다.

여기까지 보안 클라이언트에서 시작되는 ″아웃바운드″ TCP 연결에 대해 다루었습니다. 클라이언트 SOCKS 지원으로 방화벽을 통해 인바운드 연결 요구를 허용하도록 SOCKS 서버에 알릴 수도 있습니다. 보안 클라이언트 시스템에서의 **Rbind()** 호출은 이러한 통신을 허용합니다. 작동할 **Rbind()**의 경우 보안 클라이언트는 이전에 **connect()** 호출을 발행했어야 하며 호출은 SOCKS 서버를 통한 아웃바운드 연결이 이루어져야 합니다. **Rbind()** 인바운드 연결은 **connect()**가 설정한 아웃바운드 연결에서 사용된 동일한 IP 주소에서 와야 합니다.

다음 그림은 소켓 함수가 어플리케이션에 투명한 SOCKS 서버와 상호작용하는 방법에 대한 자세한 개요를 보여줍니다. 예에서 FTP 클라이언트는 **bind()** 함수[1] 대신에 **Rbind()** 함수를 호출합니다. **bind()**를 **Rbind()**가 되도록 정의하는 __Rbind preprocessor #define으로 FTP 클라이언트 코드를 다시 컴파일하여 이 호출을 수행합니다. 혹은 어플리케이션이 명시적으로 관계있는 소스 코드에 **Rbind()**를 코드화할 수 있습니다. 어플리케이션에서 SOCKS 서버를 통해 인바운드 연결을 필요로 하지 않는 경우 **Rbind()**를 사용해서는 안됩니다.

주:

1. FTP 프로토콜을 사용하여 FTP 서버가 파일이나 자료를 송신할 FTP 클라이언트의 요청으로 인해 자료 연결을 설정할 수 있으므로 FTP 클라이언트는 **Rbind()**를 사용합니다.
2. SOCKS 서버는 FTP 클라이언트와 자료 연결을 설정하고 FTP 클라이언트와 FTP 서버 사이에 자료를 전달합니다. 많은 SOCKS 서버를 사용하여 서버의 고정 길이 시간은 보안 클라이언트에 연결될 수 있습니다. 이 시간 내에 서버가 연결되지 않으면 errno ECONNABORTED가 **accept()**에서 발생합니다.
3. FTP 클라이언트는 SOCKS 서버를 통해 비보안 네트워크로의 아웃바운드 TCP 연결을 초기설정합니다. FTP 클라이언트가 connect에 지정한 목적지 주소는 비보안 네트워크에 위치한 FTP 서버의 IP 주소 및 포트입니다. 보안 호스트 시스템은 이러한 연결이 SOCKS 서버를 통해 지정되도록 SOCKS 페이지를 통해 구성되었습니다. 일단 구성되면 시스템에서는 SOCKS 페이지를 통해 지정된 SOCKS 서버로 연결을 자동으로 지정합니다.
4. 소켓이 열리며 **Rbind()**는 인바운드 TCP 연결을 설정하기 위해 호출됩니다. 일단 설정되면 이 인바운드 연결은 위에 지정된 동일한 목적지 아웃바운드 IP 주소에서 옵니다. SOCKS 서버를 통한 아웃바운드 및 인바운드 연결은 특정 스레드에 대해 쌍을 이루어야 합니다. 반면, 모든 **Rbind()** 인바운드 연결은 SOCKS 서버를 통한 아웃바운드 연결 바로 다음에 와야 하며 이 스레드와 관련된 비SOCKS 연결 개입은 **Rbind()**를 실행하기 전에 시도될 수 없습니다.
5. **getsockname()**은 SOCKS 서버 주소를 리턴합니다. 소켓은 SOCKS 서버를 통해 선택된 포트와 쌍을 이루는 SOCKS 서버 IP 주소와 논리적으로 바인드됩니다. 이 예에서 주소는 ″제어 연결″ 소켓 CTLed를 통해 비보안 네트워크에 위치한 FTP 서버로 송신됩니다. 이것은 FTP 서버가 연결된 주소입니다. FTP 서버는 SOCKS 서버로 연결되고 보안 호스트에는 직접 연결되지 않습니다.

6. SOCKS 서버는 FTP 클라이언트와 자료 연결을 설정하고 FTP 클라이언트와 FTP 서버 사이에 자료를 전달합니다. 많은 SOCKS 서버를 사용하여 서버의 고정 길이 시간은 보안 클라이언트에 연결될 수 있습니다. 이 시간 내에 서버가 연결되지 않으면 errno ECONNABORTED가 **accept()**에서 발생합니다.

스레드 안전성

동일한 프로세스 내에서 복수 스레드에서 동시에 시작할 수 있는 경우 함수는 스레드세이프로 간주됩니다. 호출된 모든 함수가 스레드세이프인 경우에도 함수는 스레드세이프입니다. 소켓 API는 둘 모두 스레드세이프인 시스템 및 네트워크 함수로 구성됩니다.

이름이 "_r"로 끝나는 모든 네트워크 함수는 유사한 의미론을 가지므로 또한 스레드세이프입니다. 스레드 세이프 소켓 API를 사용하는 샘플 프로그램은 예: 스레드 세이프 네트워크 루틴에 gethostbyaddr_r() 사용을 참조하십시오.

기타 분석기 루틴은 서로에 대해 스레드세이프이나 _res 자료 구조를 사용합니다. 이 자료 구조는 프로세스에서 모든 스레드 사이에 공유되며 분석기 호출시 어플리케이션에서 변경될 수 있습니다. 분석기 루틴을 사용하는 샘플 프로그램은 예: DNS 갱신 및 조회를 참조하십시오.

비블록화 I/O

어플리케이션이 소켓 입력 함수 중 하나를 발행하고 읽을 자료가 없는 경우 함수는 블록화되고 읽을 자료가 있을 때까지 리턴되지 않습니다. 마찬가지로 어플리케이션은 즉시 자료를 송신할 수 없을 때 소켓 출력 함수를 블록화할 수 있습니다. 마지막으로, **connect()** 및 **accept()**는 상대의 프로그램과의 연결 설정을 기다리는 동안 블록화될 수 있습니다.

소켓은 지연 없이 함수가 리턴하도록 어플리케이션 프로그램이 블록화하는 함수를 발행할 수 있는 메소드를 제공합니다. 이는 **fcntl()**을 호출하여 **O_NONBLOCK** 플래그를 켜거나 **ioctl()**을 호출하여 **FIONBIO** 플래그를 켜서 수행됩니다. 일단 이 비블록화 모드로 실행되면 함수가 블록화 없이 완료될 수 없는 경우 바로 리턴됩니다. **connect()**는 연결 시작이 시작되었다는 것을 의미하는 **[EINPROGRESS]**와 함께 리턴될 수 있습니다. 그런 다음, **select()**를 사용하여 연결이 완료된 시점을 판별할 수 있습니다. 비블록화 모드로 실행하여 영향을 받는 다른 모든 함수에 경우 **[EWOULDBLOCK]** 오류 코드는 호출에 실패했다는 것을 나타냅니다.

다음 소켓 함수로 비블록화를 사용할 수 있습니다.

- **accept()**
- **connect()**
- **gsk_secure_soc_read()**
- **gsk_secure_soc_write()**
- **read()**
- **readv()**
- **recv()**
- **recvfrom()**

- **recvmsg()**
- **send()**
- **send_file()**
- **send_file64()**
- **sendmsg()**
- **sendto()**
- **SSL_Read()**
- **SSL_Write()**
- **write()**
- **writev()**

비블록화 I/O를 사용하는 샘플 프로그램은 예: 비블록화 I/O 및 select()를 참조하십시오.

신호

어플리케이션 프로그램은 어플리케이션이 관련된 상황이 발생할 때 비동기적으로 통지되도록 요구(시스템에서 **신호**를 송신하도록 요구)할 수 있습니다. 소켓이 어플리케이션으로 송신하게 될 비동기 신호에는 두 가지가 있습니다.

1. **SIGURG**는 대역폭을 벗어난(OOB) 자료가 OOB 자료 개념이 지원되는 소켓에서 수신될 때 송신될 신호입니다. 예를 들어, 주소 패밀리가 AF_INET이고 유형이 SOCK_STREAM인 소켓은 SIGURG 신호를 송신하는 것을 조건으로 할 수 있습니다.
2. **SIGIO**는 정상 자료, OOB 자료, 오류 조건 또는 모든 것이 어떤 유형의 소켓에 발생할 때 송신될 신호입니다.

어플리케이션은 시스템이 신호를 송신하도록 요구하기 전에 신호 수신을 처리할 수 있도록 해야 합니다. 이는 **신호 핸들러**를 설정하여 수행됩니다. 신호 핸들러를 설정하는 한 가지 방법은 **sigaction()** 호출을 발행하는 것입니다.

어플리케이션은 다음과 같은 방법으로 시스템이 **SIGURG** 신호를 송신하도록 요구합니다.

- **fcntl()** 호출을 발행하고 F_SETOWN 명령으로 프로세스 ID나 프로세스 그룹 ID 지정.
- **ioctl()** 호출 발행 및 FIOSETOWN 또는 SIOCSPGRP 명령(요구) 지정.

어플리케이션은 시스템이 두 단계로 **SIGIO** 신호를 송신하도록 요구합니다. 우선, **SIGURG** 신호에 대해 위에 설명한 바와 같이 프로세스 그룹 ID 또는 프로세스 ID를 설정해야 합니다. 이는 시스템에 어플리케이션에 신호가 전달될 위치를 알리게 됩니다. 그 다음, 어플리케이션은 다음 중 한 가지를 수행해야 합니다.

- **fcntl()** 호출을 발행하고 FASYNC 플래그와 함께 F_SETFL 명령을 지정하십시오.
- **ioctl()** 호출을 발행하고 FIOASYNC 명령을 지정하십시오.

이 단계에서는 시스템이 SIGIO 신호를 생성하도록 요구합니다. 이 단계는 아무 순서로나 수행될 수 있습니다. 또한, 어플리케이션이 이러한 요구를 청취 소켓에서 발행할 경우 요구에 의해 설정된 값은 **accept()** 함수에서

어플리케이션으로 리턴되는 모든 소켓에서 계승됩니다. 다시 말해, 새로 승인된 소켓도 SIGIO 신호를 보내는 데 있어서 동일한 정보와 마찬가지로 동일한 프로세스 ID 또는 프로세스 그룹 ID를 갖게 됩니다.

또한, 소켓은 오류 조건 발생시 동시 신호를 생성할 수 있습니다. 어플리케이션이 소켓 함수에서 [EPIPE]를 **errno**로서 수신하는 경우 SIGPIPE 신호는 **errno** 값을 수신하는 조작을 발행한 프로세스로 전달됩니다. BSD 구현에서 기본적으로 SIGPIPE 신호는 **errno** 값을 수신한 프로세스를 종료합니다. 이전 OS/400 구현 릴리스와 호환 가능한 상태로 두기 위해 OS/400 구현은 SIGPIPE 신호에 대해 무시의 디폴트 작동을 사용합니다. 이는 기존의 어플리케이션이 신호 함수 추가에 의해 부정적인 영향을 받지 않게 합니다.

소켓 함수에서 블록화된 프로세스로 신호가 전달될 때 함수가 어플리케이션의 신호 핸들러가 실행될 수 있는 **[EINTR] errno** 값과 함께 wait에서 리턴됩니다. 이러한 상황이 발생할 함수는 다음과 같습니다.

- **accept()**
- **connect()**
- **read()**
- **readv()**
- **recv()**
- **recvfrom()**
- **recvmsg()**
- **select()**
- **send()**
- **sendto()**
- **sendmsg()**
- **write()**
- **writev()**

신호가 신호를 보내는 조건이 실제로 존재하는 위치를 식별하는 소켓 설명자를 어플리케이션 프로그램에 제공하지 않는다는 점에 유의하는 것이 중요합니다. 따라서 어플리케이션 프로그램이 여러 소켓 설명자를 사용하고 있는 경우 설명자를 폴링하거나 **select()** 호출을 사용하여 신호가 수신된 이유를 판별해야 합니다.

신호를 사용하는 샘플 프로그램은 예: 소켓 API 블록화에 신호 사용을 참조하십시오.

IP 멀티캐스팅

IP 멀티캐스팅을 사용하여 어플리케이션이 네트워크의 호스트 그룹이 수신할 수 있는 단일 IP 데이터그램을 송신할 수 있습니다. 그룹에 있는 호스트는 단일 서브네트에 있을 수 있거나 멀티캐스트 가능 라우터를 연결하는 다른 서브네트에 있을 수도 있습니다. 호스트는 언제라도 그룹에 가입하거나 그룹에서 탈퇴할 수 있습니다. 호스트 그룹 내에서는 멤버 번호나 위치에 제한이 없습니다. 범위 224.0.0.1에서 239.255.255.255까지의 클래스 D 인터넷 주소는 호스트 그룹을 식별합니다.

현재 AF_INET 주소 패밀리에만 IP 멀티캐스팅을 사용할 수 있습니다.

어플리케이션 프로그램은 소켓 API 및 무접속 SOCK_DGRAM 유형 소켓을 사용하여 멀티캐스트 데이터그램을 송신하거나 수신할 수 있습니다. 멀티캐스팅은 일 대 다(one-to-many) 전송 방법입니다. SOCK_STREAM 유형의 연결 지향 소켓은 멀티캐스팅에 사용될 수 있습니다. 소켓 유형이 SOCK_DGRAM으로 작성되면 어플리케이션은 **setsockopt()** 함수를 사용하여 해당 소켓과 관련된 멀티캐스트 특성을 제어할 수 있습니다. **setsockopt()** 함수는 다음 IPPROTO_IP 레벨 플래그를 허용합니다.

- IP_ADD_MEMBERSHIP: 지정된 멀티캐스트 그룹에 가입
- IP_DROP_MEMBERSHIP: 지정된 멀티캐스트 그룹에서 탈퇴
- IP_MULTICAST_IF: 발신 멀티캐스트 데이터그램이 송신될 인터페이스 설정
- IP_MULTICAST_TTL: 발신 멀티캐스트 데이터그램에 대해 IP 헤더에 TTL(생존 시간) 설정
- IP_MULTICAST_LOOP: 멀티캐스트 그룹 멤버로 있는 한 발신 멀티캐스트 데이터그램 사본을 송신 호스트로 전달해야 하는지 여부를 지정

IP 멀티캐스트의 예는 예: 멀티캐스트 사용을 참조하십시오.

파일 자료 전송--send_file() 및 accept_and_recv()

OS/400 소켓에서는 연결된 소켓을 통해 더 빠르고 더 쉽게 파일을 전송할 수 있게 하는 **send_file()** 및 **accept_and_recv()** API를 제공합니다. 이러한 두 API는 HTTP(하이퍼텍스트 전송 프로토콜) 서버와 같은 파일 제공 어플리케이션에 특히 유용합니다.

send_file()로 단일 API 호출을 사용하여 연결된 소켓상의 파일 시스템에서 직접 파일 자료를 송신할 수 있습니다.

accept_and_recv()는 세 가지의 소켓 함수 **accept()**, **getsockname()**, **recv()**를 조합한 것입니다.

send_file() 및 **accept_and_recv()** API의 샘플 프로그램은 예: send_file() 및 accept_and_recv()를 사용하여 파일 자료 전송을 참조하십시오.

대역폭을 벗어난 자료

대역폭을 벗어나는(OOB) 자료란 연결 지향(스트림) 소켓에 대해서만 의미를 갖는 사용자별 자료를 말합니다. 일반적으로 스트림 자료는 송신된 순서와 동일한 순서로 수신됩니다. OOB 자료는 스트림 내의 위치와 독립적으로(송신된 순서와 독립적으로) 수신됩니다. A 프로그램에서 B 프로그램으로 송신될 때 B 프로그램에 도착이 통지되는 방식으로 자료가 표시되므로 이것이 가능합니다.

| OOB 자료는 AF_INET(SOCK_STREAM) 및 AF_INET6(SOCK_STREAM)에서만 지원됩니다.

OOB 자료는 **send()**, **sendto()** 및 **sendmsg()** 함수에 MSG_OOB 플래그를 지정함으로써 송신됩니다.

OOB 자료의 전송은 일반 자료의 전송과 같습니다. 버퍼된 자료 이후에 송신됩니다. 다시 말해, OOB 자료는 버퍼링될 수 있는 자료에 우선권을 갖지 못합니다. 자료는 송신된 순서대로 전송됩니다.

수신측의 경우 제반 상황이 다소 더 복잡합니다.

- 소켓 API가 OOB 마커를 사용하여 시스템상에서 수신되는 OOB 자료를 추적합니다. OOB 마커는 송신된 OOB 자료의 마지막 바이트를 가리킵니다.

  주: OOB 마커가 어떤 바이트를 가리키는지를 표시하는 값은 시스템에 따라 설정됩니다. (모든 어플리케이션이 해당 값을 사용함)이 값은 TCP 연결의 로컬 종단과 리모트 종단 사이에 일치해야 합니다. 이 값을 사용하는 소켓 어플리케이션은 클라이언트 및 서버 어플리케이션 사이에서 이를 일관성있게 사용해야 합니다. OOB 마커가 나타내는 바이트를 변경하려면 Information Center에서 CHGTCPA(TCP 속성 변경) 명령 변경을 참조하십시오.

  SIOCATMARK ioctl() 요구는 읽기 포인터가 마지막 OOB 바이트를 가리키는지를 판별합니다.

  주: OOB 자료가 여러 번 발생하여 송신되는 경우 OOB 마커는 마지막 OOB 자료 발생의 마지막 OOB 바이트를 가리킵니다.
- OOB 자료를 인라인 수신하는지 여부에 관계 없이, OOB 자료가 송신된 경우 입력 조작은 OOB 마커까지 자료를 처리합니다.
- **recv()**, **recvmsg()** 또는 **recvfrom()** 함수(MSG_OOB 플래그가 설정된)는 OOB 자료를 수신하는 데 사용됩니다. 수신 함수 중 하나가 완료되고 다음 중 한 가지가 발생할 경우 [EINVAL] 오류가 리턴됩니다.
  - SO_OOBINLINE 소켓 옵션이 설정되어 있지 않고 수신할 OOB 자료가 없습니다.
  - SO_OOBINLINE 소켓 옵션이 설정되어 있습니다.

  SO_OOBINLINE 소켓 옵션이 설정되어 있지 않고 송신 프로그램에서 1바이트보다 큰 크기의 OOB 자료를 송신했으면 마지막 바이트를 제외한 모든 바이트가 정상 자료로 간주됩니다(일반 자료란 수신 프로그램이 MSG_OOB 플래그를 지정하지 않고 자료를 수신할 수 있음을 의미합니다). 송신된 OOB 자료의 마지막 바이트는 정상 자료 스트림에 저장되지 않습니다. 이 바이트는 MSG_OOB 플래그가 설정된 **recv()**, **recvmsg()** 또는 **recvfrom()** 함수를 발행해야만 검색할 수 있습니다. MSG_OOB 플래그가 설정되지 않은 수신이 발행되어 정상 자료가 수신되면 OOB 바이트는 삭제됩니다. 또한, OOB 자료가 여러 번 발생하여 송신되면 이전 발생의 OOB 자료는 유실되며 최종 OOB 자료 발생의 OOB 자료 위치가 기억됩니다.

  SO_OOBINLINE 소켓 옵션이 설정되면 송신된 모든 OOB 자료는 정상 자료 스트림에 저장됩니다. MSG_OOB 플래그를 지정하지 않고(이 플래그가 지정되어 있는 경우 [EINVAL] 오류가 리턴됩니다) 세 가지 수신 함수 중 하나를 발행하여 자료를 검색할 수 있습니다. OOB 자료는 OOB 자료가 여러 번 발생하여 송신되는 경우 유실되지 않습니다.
- OOB 자료는 SO_OOBINLINE이 설정되어 있지 않은 경우에 삭제되며 OOB 자료가 수신된 후에는 사용자가 SO_OOBINLINE을 on으로 설정합니다. 초기 OOB 바이트는 정상 자료로 간주됩니다.
- SO_OOBINLINE이 설정되어 있지 않고 OOB 자료가 송신된 후 수신 프로그램에서 OOB 자료를 수신하기 위해 입력 함수를 발행한 경우 OOB 마커는 여전히 유효합니다. OOB 바이트가 수신된 경우에도 수신 프로그램에서는 여전히 읽기 포인터가 OOB 마커에 있는지 검사할 수 있습니다.

I/O 멀티플렉싱--select()

비동기 I/O에서 어플리케이션 자원을 최대화하기 위해 보다 더 효율적인 방법을 제공하므로, **select()** API가 아닌 비동기 I/O API를 사용하는 것이 바람직합니다. 그러나 특정 어플리케이션 설계에서는 **select()**를 사용할 수 있습니다. 비동기 I/O와 마찬가지로, **select()**는 동시에 여러 조건에서 대기할 공통 지점을 작성합니다. 그러나 **select()**를 사용하여 어플리케이션은 다음을 수행하기 위해 설명자 세트를 지정할 수 있습니다.

- 읽혀질 자료가 있는지 판별.
- 자료가 기록될 수 있는지 판별.
- 예외 조건이 존재하는지 판별.

각 세트에 지정할 수 있는 설명자는 소켓 설명자, 파일 설명자 또는 설명자로 표시되는 기타 모든 오브젝트일 수 있습니다.

또한 **select()** 함수는 어플리케이션이 자료를 사용할 수 있을 때까지 대기할 것인지를 지정할 수 있습니다. 어플리케이션이 대기할 시간을 지정할 수 있습니다. 샘플 프로그램은 예: 비블록화 I/O 및 select()를 참조하십시오.

소켓 네트워크 함수

소켓 네트워크 함수는 어플리케이션 프로그램이 호스트, 프로토콜, 서비스 및 네트워크 파일에서 정보를 얻을 수 있도록 합니다. 파일의 이름, 주소 또는 순차 액세스별로 정보에 액세스할 수 있습니다. 이러한 네트워크 함수(또는 루틴)는 네트워크를 통해 실행되는 프로그램 사이에서 통신 설정시에 필요하므로 AF_UNIX 소켓에서 사용되지 않습니다. 이러한 네트워크 함수 루틴 각각의 간략한 요약은 Information Center의 API 참조 주제에서 소켓 네트워크 함수(루틴)를 참조하십시오.

루틴은 다음을 수행합니다.

- 호스트명을 네트워크 주소로 맵핑합니다.
- 네트워크명을 네트워크 번호로 맵핑합니다.
- 프로토콜명을 프로토콜 번호로 맵핑합니다.
- 서비스명을 포트 번호로 맵핑합니다.
- 인터넷 네트워크 주소의 바이트 순서를 변환합니다.
- 인터넷 주소 및 점 십진 표기법을 변환합니다.

네트워크 루틴에는 분석기(resolver) 루틴이라는 루틴 그룹이 포함되어 있습니다. 이러한 루틴은 인터넷 정의역에서 이름 서버에 대한 패킷을 작성, 송신 및 해석하고 이름 분석을 수행하는 데에도 사용됩니다. 일반적으로 분석기 루틴은 **gethostbyname()**, **gethostbyaddr()**, **getnameinfo()** 및 **getaddrinfo()**에 의해 호출되지만 직접 호출될 수 있습니다. 이러한 분석기 루틴을 사용하는 예는 예: 스레드세이프 네트워크 루틴에 gethostbyaddr_r() 사용을 참조하십시오. 주로 분석기 루틴은 소켓 어플리케이션을 통해 정의역명 시스템(DNS)에 액세스하는 데 사용됩니다. 소켓이 DNS와 함께 사용되는 방법에 대한 자세한 내용은 정의역명 서비스 지원을 참조하십시오.

정의역명 시스템(DNS) 지원

iSeries는 분석기 기능을 통해 정의역명 시스템(DNS)에 대한 액세스를 어플리케이션에 제공합니다. DNS에는 다음과 같은 세 가지 주요 구성요소가 있습니다.

- **정의역명 공간 및 자원 레코드**
  트리 구조의 이름 공간과 이 이름과 연관된 자료에 대한 스펙.
- **이름 서버**
  정의역 트리 구조에 대한 정보를 보유하고 정보를 설정하는 서버 프로그램. 이름 서버에 대한 자세한 정보는 Information Center의 DNS 주제를 참조하십시오.
- **분석기**
  클라이언트 요구에 응답시 이름 서버에서 정보를 추출하는 프로그램.

OS/400 구현에서 제공되는 분석기는 이름 서버와의 통신을 제공하는 소켓 기능입니다. 이러한 루틴은 패킷을 작성, 송신, 갱신 및 해석한 후 성능을 위한 이름 캐싱을 수행하는 데 사용할 수 있습니다. 이들은 또한 ASCII에서 EBCDIC으로, EBCDIC에서 ASCII으로의 변환 함수를 제공합니다. 선택적으로, 분석기는 DNS와 안전하게 통신하기 위해 트랜잭션 서명(TSIG)을 사용합니다. 개별 분석기 루틴의 간략한 요약은 Information Center의 API 참조 주제에서 소켓 네트워크 함수(루틴)를 참조하십시오. 또한 이 링크는 _res 구조의 정보를 제공합니다. _res에는 이러한 루틴에서 사용되는 글로벌 정보가 포함됩니다.

정의역명에 대한 자세한 정보는 RFC 탐색 페이지에서 찾을 수 있는 다음 RFC를 참조하십시오.

- RFC 1034, ″정의역명 - 개념 및 기능″
- RFC 1035, ″정의역명 - 구현 및 스펙″
- RFC 1886, ″IP 버전 6을 지원하기 위한 DNS 확장 기능
- RFC 2136, ″정의역명 시스템의 동적 갱신(DNS UPDATE)″
- RFC 2181, ″DNS 스펙의 설명″
- RFC 2845, ″DNS용 비밀 키 트랜잭션 인증(TSIG)″
- RFC 3152, ″IP6.ARPA에 대한 DNS 위임″

| 소켓 어플리케이션과 DNS를 함께 사용하는 기타 방법에 대한 자세한 정보는 다음 주제를 참조하십시오.

| • 환경 변수
| 이 주제에서는 이름 분석에 사용할 수 있는 환경 변수를 설명합니다.
| • 자료 캐싱
| 이 주제에는 네트워크에서 통신량을 줄이기 위해 소켓을 사용하여 DNS 조회에 대한 응답을 캐시하는 것에 대한 세부사항이 있습니다. 예: DNS 갱신 및 조회에서는 소켓 API를 사용하여 DNS 레코드를 조회하고 갱신하는 방법을 나타내는 샘플 프로그램을 제공합니다.

환경 변수

환경 변수를 사용하여 분석기 기능에 대한 디폴트 초기화를 대체할 수 있습니다. 환경 변수는 **res_init()** 또는 **res_ninit()** 호출이 성공한 후에만 검사됩니다. 따라서 수동으로 구조가 초기화된 경우 환경 변수는 무시됩니다. 구조는 한 번만 초기화되므로 이후의 환경 변수 변경사항은 무시된다는 점도 참고하십시오.

주: 환경 변수명은 반드시 대문자여야 합니다. 스트링 값은 대소문자를 혼합할 수 있습니다. CCSID 290을 사용하는 일본어 시스템에서는 환경 변수명과 값 모두에서 대문자와 숫자만 사용해야 합니다. 리스트에는 **res_init()** 및 **res_ninit()** API와 함께 사용할 수 있는 환경 변수에 대한 설명이 포함되어 있습니다.

LOCALDOMAIN

탐색 정의역이 최대 6개로 총 256자(공백 포함)인 공백으로 분리된 리스트에 이 환경 변수를 설정하십시오. 이것은 구성된 탐색 리스트(struct state.defdname 및 struct state.dnsrch)를 대체합니다. 탐색 리스트가 지정된 경우 디폴트 로컬 정의역은 조회에서 사용되지 않습니다.

RES_OPTIONS

특정 내부 분석기 변수를 수정할 수 있도록 합니다. 공백으로 분리된 다음 옵션 중 하나 이상에 이 환경 변수를 설정할 수 있습니다.

- **NDOTS:n**
  초기 절대 조회가 이루어지기 전에 **res_query()**에 부여된 이름에 나타나야 하는 도트 수에 대한 임계값을 설정합니다. n의 디폴트 값은 “1”로 이름에 도트가 있는 경우 이름에 탐색 리스트 요소가 추가되기 전에 그 이름을 절대 이름으로 먼저 시도함을 의미합니다.
- **TIMEOUT:n**
  포기하고 조회를 재시도하기 전에 분석기가 리모트명 서버로부터 응답을 기다리는 시간(초 단위)을 설정합니다.
- **ATTEMPTS:n**
  포기하고 나열된 다음 이름 서버를 시도하기 전에 분석기가 주어진 이름 서버로 송신할 조회 수를 설정합니다.
- **ROTATE**
  _res.options에서 RES_ROTATE를 설정하는데 나열된 이름 서버 중에서 이름 서버 선택을 순환시킵니다. 이렇게 하면 모든 클라이언트가 매번 첫 번째 나열된 서버를 가장 먼저 시도하게 되지 않고 나열된 모든 서버간에 조회 로드가 스프레드됩니다.
- **NO-CHECK-NAMES**
  _res.options에서 RES_NOCHECKNAME을 설정하는 데 수신 호스트명 및 메일명에 유효하지 않은 문자(예: 밑줄 (_), 비ASCII 또는 제어 문자)가 있는지에 대한 최신 BIND 검사를 작동 불가능하게 합니다.

QIBM_BIND_RESOLVER_FLAGS

이 환경 변수를 공백으로 분리된 분석기 옵션 플래그 리스트에 설정하십시오. 이것은 RES_DEFAULT 옵션(struct state.options) 및 시스템 구성 값(CHGTCPDMN - TCP/IP 정의역 변경)을 대체합니다. state.options 구조는 일반적으로 RES_DEFAULT, OPTIONS 환경 값 및 CHGTCPDMN 구성 값을

사용하여 초기화됩니다. 그런 다음 이 환경 변수가 사용되어 해당 디폴트 값을 대체합니다. 이 환경 변수에서 명명된 플래그 앞에 '+', '-' 또는 'NOT_'을 추가하여 값을 설정('+') 또는 재설정('-', 'NOT_')할 수 있습니다.

예를 들어, RES_NOCHECKNAME을 켜고 RES_ROTATE를 끄려면 문자 기반의 인터페이스에서 다음 명령을 사용하십시오.

```
ADDENVVAR ENVVAR(QIBM_BIND_RESOLVER_FLAGS) VALUE('RES_NOCHECKNAME NOT_RES_ROTATE')
```

또는

```
ADDENVVAR ENVVAR(QIBM_BIND_RESOLVER_FLAGS) VALUE('+RES_NOCHECKNAME -RES_ROTATE')
```

QIBM_BIND_RESOLVER_SORTLIST
정렬 리스트(struct state.sort_list)를 작성하려면 점 십진 형식(9.5.9.0/255.255.255.0)으로 최대 10 개의 IP 주소/마스크 쌍이 있는 공백으로 분리된 리스트에 이 환경 변수를 설정하십시오.

자료 캐싱

DNS 조회에 대한 응답 캐싱은 네트워크 통신량을 줄이기 위한 노력으로 OS/400 소켓에서 수행됩니다. 캐시는 필요에 따라 추가되고 갱신됩니다.

RES_AAONLY(인증된 응답만)가 _res.options에 설정되어 있는 경우 조회는 항상 네트워크상에서 송신됩니다. 이 경우 캐시는 응답에 대해 결코 검사하지 않습니다. RES_AAONLY가 설정되어 있지 않은 경우 네트워크상에서 이를 송신하려는 시도가 수행되기 전에 조회에 대한 응답에 대해 캐시가 검사됩니다. 응답이 있고 생존 시간이 만료되지 않은 경우 응답은 조회에 대한 응답으로 사용자에게 리턴됩니다. 생존 시간이 만료되면 항목이 제거되고 조회는 네트워크에서 송신됩니다. 또한, 캐시에 응답이 없으면 조회는 네트워크에서 송신됩니다.

응답이 인증된 경우 네트워크의 응답이 캐시에 넣어집니다. 인증되지 않은 응답은 캐시에 넣어지지 않습니다. 또한, 역조회의 결과로 수신된 응답도 캐시에 넣어지지 않습니다. CHGTCPDMN, CFGTCP 옵션 12를 사용하거나 iSeries Navigator를 통해 DNS 구성을 갱신하면 이 캐시를 지울 수 있습니다.

자료 캐싱을 사용하는 프로그램 예는 예: DNS 갱신 및 조회를 참조하십시오.

BSD(Berkeley Software Distribution) 호환

소켓은 BSD(Berkeley Software Distributions) 인터페이스입니다. 어플리케이션이 수신하는 리턴 코드와 지원되는 함수에서 사용 가능한 인수와 같은 의미론(semantics)을 BSD 의미론이라고 합니다. 그러나 일부 BSD 의미론은 OS/400 구현시 사용할 수 없으며 시스템에서 실행하기 위해서는 일반적인 BSD 소켓 어플리케이션을 변경해야 합니다.

다음 리스트에는 OS/400 구현과 BSD 구현 사이의 차이점이 요약되어 있습니다.

/etc/hosts, /etc/services, /etc/networks 및 /etc/protocols
이러한 파일의 경우 OS/400 구현은 각각 동일한 기능을 제공하는 다음과 같은 데이터베이스 파일을

제공합니다.

QUSRSYS 파일	내용
QATOCHOST	호스트명 및 해당 IP 주소 리스트
QATOCPN	네트워크 리스트와 해당 IP 주소
QATOCPP	인터넷에서 사용되는 프로토콜 리스트
QATOCPS	서비스 리스트와 서비스에서 사용하는 특정 포트 및 프로토콜.

/etc/resolv.conf

OS/400 구현시 이 정보는 iSeries Navigator에서 TCP/IP 등록 정보 페이지를 사용하여 구성되어야 합니다. TCP/IP 등록 정보 페이지에 액세스하려면 다음 단계를 완료하십시오.

1. iSeries Navigator에서 **iSeries 서버 --> 네트워크 --> TCP/IP 구성**을 펼치십시오.
2. **TCP/IP 구성**을 마우스 오른쪽 버튼으로 클릭하십시오.
3. **등록 정보**를 클릭하십시오.

bind() BSD 시스템에서 클라이언트는 socket()을 사용하여 AF_UNIX 소켓을 작성하고 connect()를 사용하여 서버에 연결한 후 bind()를 사용하여 해당 소켓에 이름을 바인드할 수 있습니다. OS/400 구현에서는 이러한 시나리오를 지원하지 않습니다(bind()가 실패합니다).

close()

OS/400 구현은 SNA를 통한 AF_INET 소켓을 제외한 close()의 링거 타이머(linger timer)를 지원합니다. 일부 BSD 구현은 close()에 대한 링거 타이머(linger timer)를 지원하지 않습니다.

connect()

BSD 시스템에서 connect()가 주소에 이전에 연결된 소켓에 대해 발행되고 무접속 전송 서비스를 사용 중이며 유효하지 않은 주소나 유효하지 않은 주소 길이가 사용되는 경우 소켓은 더 이상 연결되지 않습니다. OS/400 구현은 이러한 시나리오를 지원하지 않습니다(connect()는 실패하고 소켓은 여전히 연결되어 있습니다).

connect()가 발행된 무접속 전송 소켓은 address_length 매개변수를 0으로 설정하고 다른 connect()를 발행하여 단절될 수 있습니다.

accept(), getsockname(), getpeername(), recvfrom() 및 recvmsg()

AF_UNIX 또는 AF_UNIX_CCSID 주소 패밀리를 사용하고 소켓이 바인드되지 않았으면 디폴트 OS/400 구현은 주소 길이 0과 지정되지 않은 주소 구조를 리턴할 수있습니다. OS/400 BSD 4.4/UNIX 98 및 기타 구현은 주소 패밀리가 방금 지정된 작은 주소 구조를 리턴할 수 있습니다.

ioctl()

- 소켓 유형 SOCK_DGRAM의 BSD 시스템에서 FIONREAD 요구는 자료 길이에 주소 길이를 더한 값을 리턴합니다. OS/400 구현에서 FIONREAD는 자료 길이만 리턴합니다.
- 대부분의 **ioctl()**의 BSD 구현에서 사용할 수 있는 모든 요구가 **ioctl()**의 OS/400 구현에서 사용 가능한 것은 아닙니다.

listen()

BSD 시스템에서 0보다 작은 값으로 백로그 매개변수가 설정된 **listen()**을 발행해도 오류가 발생하지 않습니다. 또한, 경우에 따라 BSD 구현은 백로그 매개변수를 사용하지 않거나 알고리즘을 사용하여 백로그 값의 최종 결과를 제안합니다. OS/400 구현은 백로그 값이 0보다 작은 경우 오류를 리턴합니다. 백로그를 유효한 값으로 설정하면 값은 백로그로 사용됩니다. 그러나 백로그를 {SOMAXXCONN} 보다 큰 값으로 설정하면 백로그의 디폴트는 {SOMAXXCONN}에 설정된 값입니다.

대역폭을 벗어난(OOB) 자료

OS/400 구현에서 OOB 자료는 SO_OOBINLINE이 설정되지 않은 경우에 삭제되며 OOB 자료가 수신된 후에는 사용자가 SO_OOBINLINE을 on으로 설정합니다. 초기 OOB 바이트는 정상 자료로 간주됩니다.

socket()의 프로토콜 매개변수

추가 보안을 제공하는 수단으로, 어떠한 사용자도 방법 IPPROTO_TCP 또는 IPPROTO_UDP 프로토콜을 지정하여 SOCK_RAW 소켓을 작성할 수 없습니다.

res_xlate() 및 res_close()

이러한 함수는 OS/400 구현을 위한 분석기 루틴에 포함됩니다. **res_xlate()**는 DNS 패킷을 EBCDIC에서 ASCII로, ASCII에서 EBCDIC으로 변환합니다. **res_close()**는 RES_STAYOPEN 옵션이 설정된 **res_send()**에서 사용된 소켓을 닫는 데 사용됩니다. 또한 _res 구조도 재설정합니다.

sendmsg() 및 recvmsg()

sendmsg() 및 **recvmsg()**의 OS/400 구현은 최고 {MSG_MAXIOVLEN} I/O 벡터를 허용합니다. BSD 구현은 {MSG_MAXIOVLEN - 1} I/O 벡터를 허용합니다.

shutdown()

OS/400 구현에서 현재 소켓 설명자에서 블록화된 출력 함수는 **shutdown()** 다음에 계획 블록화됩니다. BSD 구현시, 블록화 출력 함수는 [EPIPE] errno 값으로 종료됩니다. 마찬가지로 BSD 구현은 출력값이 0인 입력 조작이 블록화 중이고 **shutdown()**이 다른 프로세스나 스레드에서 발행될 경우에 입력 조작 블록화를 끝냅니다. OS/400 구현은 단순히 출력값이 0인 후속적인 입력 함수에 실패하나 자료가 수신되거나 자료를 대기 상태에서 제거하려는 일부 다른 조치가 취해질 때까지 입력 함수의 블록화는 계속됩니다.

신호 신호 지원과 관련된 몇 가지 차이점은 다음과 같습니다.

- BSD 구현은 출력 조작시 송신되는 자료에 대해 수신확인을 수신할 때마다 SIGIO 신호를 발행합니다. OS/400 소켓 구현은 아웃바운드 자료와 관련된 신호를 생성하지 않습니다.
- SIGPIPE 신호에 대한 디폴트 조치는 BSD 구현의 프로세스를 종료하는 것입니다. 이전 릴리스의 OS/400과의 하향 호환성을 유지보수하기 위해, OS/400 구현에서는 SIGPIPE 신호에 대한 디폴트 조치인 무시를 사용합니다.

SO_REUSEADDR 옵션

BSD 시스템에서 소켓 패밀리가 AF_INET이고 유형이 SOCK_DGRAM인 소켓은 시스템이 소켓이 **connect()**에 지정된 주소에 도착하는 데 사용되는 인터페이스의 주소를 변경하도록 합니다. 예를 들어, 소켓 유형 SOCK_DGRAM을 INADDR_ANY 주소에 바인드한 후 이를 주소 a.b.c.d에 연결하

면 시스템은 사용자 소켓을 변경하므로, 주소 a.b.c.d에 패킷을 라우트하도록 선택한 인터페이스의 IP 주소에 지금 바인드됩니다. 또한, 소켓이 바인드된 이 IP 주소가 a.b.c.e에 바인드되면 이제 주소 a.b.c.e가 INADDR_ANY 대신에 **getsockname()**에 나타나며 SO_REUSEADDR 옵션은 주소가 a.b.c.e인 동일한 포트 번호에 또 다른 소켓을 바인드하는 데 사용되어야 합니다.

반대로, 이 예에서 OS/400 구현은 로컬 주소를 INADDR_ANY에서 a.b.c.e로 변경하지 않습니다. **getsockname()**은 연결이 수행된 후에 계속해서 INADDR_ANY를 리턴합니다.

SO_SNDBUF 및 SO_RCVBUF 옵션

BSD 시스템의 SO_SNDBUF 및 SO_RCVBUF에 설정된 값은 OS/400 구현에서보다 큰 제어 레벨을 제공합니다. OS/400 구현에서 이러한 값은 보고 값으로 처리됩니다.

UNIX 98 호환성

개발자와 벤더들의 컨소시엄인 Open Group이 작성한 UNIX 98은 UNIX가 알려진 인터넷 관련 기능을 상당수 통합하면서 UNIX 오퍼레이팅 시스템의 상호운영성을 향상시켰습니다. V5R2에서 새로운 사항으로 OS/400 소켓은 UNIX 98 오퍼레이팅 환경과 호환되는 소켓 어플리케이션을 작성할 수 있는 능력을 프로그래머에게 제공합니다. 현재 IBM은 대부분의 소켓 API의 두 가지 버전을 지원합니다. 기본 OS/400 API는 BSD 4.3 구조 및 구문을 사용합니다. 다른 버전은 BSD 4.4 및 UNIX 98 프로그래밍 인터페이스 스펙과 호환 가능한 구문 및 구조를 사용합니다. _XOPEN_SOURCE 매크로 값을 520 이상으로 정의하면 UNIX 98 호환 인터페이스를 선택할 수 있습니다.

UNIX 98 호환 어플리케이션의 주소 구조 차이점

_XOPEN_OPEN 매크로를 지정할 때 디폴트 OS/400 구현에서 사용되는 것과 동일한 주소 패밀리를 사용하여 UNIX 98 호환 어플리케이션을 작성할 수 있지만 **sockaddr** 주소 구조에 차이가 있습니다. 아래 표는 BSD 4.3 **sockaddr** 주소 구조와 UNIX 98 호환 주소 구조를 비교한 것입니다.

표 17. BSD 4.3 및UNIX 98/BSD 4.4 소켓 주소 구조 비교

BSD 4.3 구조	**BSD 4.4/UNIX 98** 호환 구조
소켓 주소 구조	

`struct sockaddr {`
`  u_short sa_family;`
`char  sa_data[14];`
`};`
	`struct sockaddr {`
	`uint8_t  sa_len;`
	`sa_family_t sa_family`
	`char  sa_data[14];`
	`};`

sockaddr_in address structure	

`struct sockaddr_in {`
`short  sin_family;`
`u_short sin_port;`
`struct  in_addr sin_addr;`
`char sin_zero[8];`
`};`
	`struct sockaddr_in {`
	`uint8_t  sin_len;`
	`sa_family_t  sin_family;`
	`u_short sin_port;`
	`struct  in_addr sin_addr;`
	`char sin_zero[8];`
	`};`

sockaddr_in6 주소 구조	

표 17. BSD 4.3 및UNIX 98/BSD 4.4 소켓 주소 구조 비교 (계속)

struct sockaddr_in6 {
sa_family_t sin6_family;
in_port_t sin6_port;
uint32_t sin6_flowinfo;
struct in6_addr sin6_addr;
uint32_t sin6_scope_id;
};
	struct sockaddr_in6 {
	uint8_t sin6_len;
	sa_family_t sin6_family;
	in_port_t sin6_port;
	uint32_t sin6_flowinfo;
	struct in6_addr sin6_addr;
	uint32_t sin6_scope_id;
	};

sockaddr_un 주소 구조	

struct sockaddr_un {
short sun_family;
char sun_path[126];
};
	struct sockaddr_un {
	uint8_t sun_len;
	sa_family_t sun_family;
	char sun_path[126]
	};

API 차이점

ILE 기반의 언어로 개발하거나 _XOPEN_SOURCE 매크로를 사용하여 어플리케이션을 컴파일할 때 일부 소켓 API는 내부 이름에 맵핑됩니다. 이런 내부 이름은 원래 API와 동일한 기능을 제공합니다. 표에는 영향을 받는 이런 API가 나열되어 있습니다. 일부 다른 C 기반 언어로 소켓 어플리케이션을 작성하고 있는 경우 이런 API의 내부 이름에 직접 작성할 수 있습니다. 이런 API 두 버전 모두에 대한 사용법 노트와 세부사항을 참조하려면 원래 API로의 링크를 사용하십시오.

표 18. API 및 UNIX 98 해당 이름

API명	내부 이름
accept()	qso_accept98()
accept_and_recv()	qso_accept_and_recv98()
bind()	qso_bind98()
connect()	qso_connect98()
endhostent()	qso_endhostent98()
endnetent()	qso_endnetent98()
endprotoent()	qso_endprotoent98()
endservent()	qso_endservent98()
getaddrinfo()	qso_getaddrinfo98()
gethostbyaddr()	qso_gethostbyaddr98()
gethostbyaddr_r()	qso_gethostbyaddr_r98()
gethostname()	qso_gethostname98()
gethostname_r()	qso_gethostname_r98()
gethostbyname()	qso_gethostbyname98()
gethostent()	qso_gethostent98()
getnameinfo()	qso_getnameinfo98()
getnetbyaddr()	qso_getnetbyaddr98()
getnetbyname()	qso_getnetbyname98()
getnetent()	qso_getnetent98()
getpeername()	qso_getpeername98()

표 18. API 및 UNIX 98 해당 이름 (계속)

getprotobyname()	qso_getprotobyname98()
getprotobynumber()	qso_getprotobynumber98()
getprotoent()	qso_getprotoent98()
getsockname()	qso_getsockname98()
getsockopt()	qso_getsockopt98()
getservbyname()	qso_getservbyname98()
getservbyport()	qso_getservbyport98()
getservent()	qso_getservent98()
inet_addr()	qso_inet_addr98()
inet_lnaof()	qso_inet_lnaof98()
inet_makeaddr()	qso_inet_makeaddr98()
inet_netof()	qso_inet_netof98()
inet_network()	qso_inet_network98()
listen()	qso_listen98()
Rbind()	qso_Rbind98()
recv()	qso_recv98()
recvfrom98()	qso_recvfrom98()
recvmsg()	qso_recvmsg98()
send()	qso_send98()
sendmsg()	qso_sendmsg98()
sendto()	qso_sendto98()
sethostent()	qso_sethostent98()
setnetent()	qso_setnetent98()
setprotoent()	qso_setprotoent98()
setservent()	qso_setprotoent98()
setsockopt()	qso_setsockopt98()
shutdown()	qso_shutdown98()
socket()	qso_socket98()
socketpair()	qso_socketpair98()

프로세스간의 설명자 전달--sendmsg() 및 recvmsg()

작업 사이의 열린설명자를 전달하는 기능은 클라이언트/서버 어플리케이션 설계의 새로운 방법을 제공할 수 있습니다. 작업 사이의 열린설명자를 전달하면 하나의 프로세스인 일반적으로 서버는 설명자를 확보하는 데 필요한 모든 작업(파일 열기, 연결 설정, **accept()** API 완료까지 대기)을 수행할 수 있으며 또 다른 프로세스인 일반적으로 작업자는 일단 설명자가 열리면 모든 자료 전송 조작을 처리할 수 있습니다. 이러한 설계는 서버와 작업자 작업 모두에 대한 논리를 보다 간단하게 합니다. 이러한 설계에서는 다른 유형의 작업자 작업이 보다 쉽게 지원될 수 있습니다. 서버는 설명자를 수신해야 할 작업자 유형을 결정하기 위해 간단한 검사만 하면 됩니다.

소켓은 서버와 작업 사이에서 설명자를 전달할 수 있는 다음의 세 가지 API 세트를 제공합니다.

- **spawn()**

  주: **spawn()**은 소켓 API가 아닙니다. OS/400 프로세스 관련 API의 일부로 제공됩니다.
- **givedescriptor()** 및 **takedescriptor()**
- **sendmsg()** 및 **recvmsg()**

spawn() API는 새로운 서버 작업(″하위 작업″이라고 함)을 시작하고 해당 하위 작업에 특정 설명자를 제공합니다. 하위 작업이 이미 활동 중인 경우 **givedescriptor()** 및 **takedescriptor()**나 **sendmsg()** 및 **recvmsg()** API를 사용해야 합니다.

그러나 **sendmsg()** 및 **recvmsg()** API는 **spawn()**, **givedescriptor()** 및 **takedescriptor()**에 비해 여러 가지 장점을 제공합니다.

이식성 **givedescriptor()** 및 **takedescriptor()** API는 표준이 아니며 iSeries에 고유합니다. iSeries와 UNIX 사이의 어플리케이션 이식성이 문제인 경우 **sendmsg()** 및 **recvmsg()** API를 대신 사용할 수 있습니다.

제어 정보의 통신

작업자 작업이 설명자를 수신할 때 다음과 같은 추가 정보를 알아야 할 경우가 있습니다.

- 설명자의 유형
- 작업자 작업이 수행해야 할 작업

sendmsg() 및 **recvmsg()** API는 사용자가 제어 정보와 같은 자료를 설명자와 함께 전송할 수 있도록 합니다. **givedescriptor()** 및 **takedescriptor()** API는 그렇지 않습니다.

성능 **sendmsg()** 및 **recvmsg()** API를 사용하는 어플리케이션은 **givedescriptor()** 및 **takedescriptor()** API를 사용하는 어플리케이션에 비해 다음 세 가지 영역에서 성능이 약간 더 좋습니다.

- 경과 시간
- CPU 이용도
- 확장성(scalability)

어플리케이션의 성능 향상 정도는 어플리케이션이 설명자를 전달하는 정도에 따라 다릅니다.

작업자 작업 풀(pool)

서버가 설명자를 전달할 수 있고 풀에 있는 작업 중 하나만 활동시켜 이를 수신하도록 작업자 작업 풀을 설정할 수 있습니다. **sendmsg()** 및 **recvmsg()** API를 사용하여 모든 작업자 작업이 공유 설명자에서 대기하도록 하여 이를 수행할 수 있습니다. 서버가 **sendmsg()**를 호출하면 작업자 작업 중 하나만이 설명자를 수신하게 됩니다.

지정되지 않은 작업자 작업 ID

givedescriptor() API에서는 서버 작업이 작업자 작업의 작업 ID를 알아야 합니다. 일반적으로 작업자 작업은 작업 ID를 취한 후 이를 자료 대기행렬과 함께 서버 작업으로 전송합니다. **sendmsg()** 및 **recvmsg()**에는 이러한 자료 대기행렬을 작성 및 관리할 별도의 오버헤드가 필요하지 않습니다.

적응 서버 설계

givedescriptor() 및 **takedescriptor()**를 사용하여 서버를 설계할 때 일반적으로 자료 대기행렬은 작업자 작업에서 서버로 작업 ID를 전송하는 데 사용됩니다. 그 다음, 서버는 **socket()**, **bind()**, **listen()**, **accept()** 등을 수행합니다. **accept()** API가 완료되면 서버는 자료 대기행렬에서 그 다음 사용 가능한 작업 ID를 수행(pull off)합니다. 그 다음, 인바운드(inbound) 연결을 해당 작업자 작업으로 전달합니다. 한 번에 너무 많은 수신 연결 요구가 발생하고 사용 가능한 작업자 작업이 충분하지 않을 경우 문제가 발생합니다. 작업자 작업 ID가 들어 있는 자료 대기행렬이 비어 있는 경우 서버는 작업자 작업이 사용 가능할 때까지 기다리는 것을 막거나 작업자 작업을 추가로 작성합니다. 여러 환경에서 이러한 대체 방법은 바람직하지 않습니다. 왜냐하면 추가 수신 요구가 청취 백로그(listen backlog)를 채울 수 있기 때문입니다.

sendmsg() 및 **recvmsg()** API를 사용하여 설명자를 전달하는 서버는 각 수신 연결을 처리할 작업자 작업을 알 필요가 없으므로 과도한 활동시 방해받지 않은 상태로 남아 있게 됩니다. 서버가 **sendmsg()**를 호출하면 수신 연결 및 제어 자료에 대한 설명자는 AF_UNIX 소켓의 내부 대기행렬에 들어가게 됩니다. 작업자 작업을 사용할 수 있게 되면 **recvmsg()**를 호출하게 되며 대기행렬에 있던 첫 번째 설명자와 제어 자료를 수신하게 됩니다.

비활동 작업자 작업

givedescriptor() API에서는 작업자 작업이 **sendmsg()** API가 사용되지 않는 동안 활동중이어야 합니다. **sendmsg()**를 호출하는 작업에는 작업자 작업에 대한 정보가 필요하지 않습니다. **sendmsg()** API에서는 AF_UNIX 소켓 연결이 설정되어야만 합니다.

sendmsg() API를 사용하여 설명자를 존재하지 않는 작업으로 전달하는 데 사용할 수 있는 방법에 대한 예는 다음과 같습니다.

서버는 **socketpair()** API를 사용하여 AF_UNIX 소켓 쌍을 작성하고 **sendmsg()** API를 사용하여 **socketpair()** 에서 작성된 AF_UNIX 소켓 중 하나를 통해 설명자를 송신한 후 **spawn()**을 호출하여 소켓 쌍의 나머지 하나를 계승하는 하위 작업을 작성할 수 있습니다. 하위 작업은 **recvmsg()**를 호출하여 서버가 전달한 설명자를 수신합니다. 하위 작업은 서버가 **sendmsg()**를 호출할 때 활동 상태가 아니었습니다.

한 번에 둘 이상의 설명자 전달

givedescriptor() 및 **takedescriptor()** API를 사용하여 한 번에 하나의 설명자만 전달할 수 있습니다. **sendmsg()** 및 **recvmsg()** API는 설명자 배열을 전달하는 데 사용할 수 있습니다.

sendmsg() 및 **recvmsg()** API를 사용하는 샘플 프로그램은 예: 프로세스간의 설명자 전달을 참조하십시오.

제 8 장 소켓 시나리오: IPv4 및 IPv6 클라이언트를 허용하기 위한 어플리케이션 작성

상황

당신이 iSeries용 소켓 어플리케이션을 전문적으로 취급하는 어플리케이션 개발 회사에 근무하는 소켓 프로그래머라고 가정하십시오. 경쟁업체보다 앞서기 위해 AF_INET6 주소 패밀리를 사용할 어플리케이션군을 개발하여 IPv4 및 IPv6으로부터의 연결을 허용하기로 결정하였습니다. IPv4 및 IPv6 노드 모두로부터 요구를 처리할 어플리케이션을 작성하려고 합니다. OS/400은 AF_INET6 주소 패밀리를 지원하여 AF_INET 주소 패밀리 소켓과의 상호운영성을 제공하는 한다는 사실을 알고 있습니다. IPv4-맵핑 IPv6 주소 형식을 사용하면 이렇게 할 수 있다는 사실도 알고 있습니다. IPv6 및 IPv4 어플리케이션간 상호운영성에 대한 작업을 하는 것에 대한 추가 정보는 IPv4 어플리케이션과의 IPv6 어플리케이션 호환성을 참조하십시오.

시나리오의 목적

이 시나리오의 목적과 목표는 다음과 같습니다.

1. IPv6 및 IPv4 클라이언트로부터 요구를 허용하고 처리하는 서버 어플리케이션 작성
2. IPv4 또는 IPv6 서버 어플리케이션으로부터 자료를 요구하는 클라이언트 어플리케이션 작성

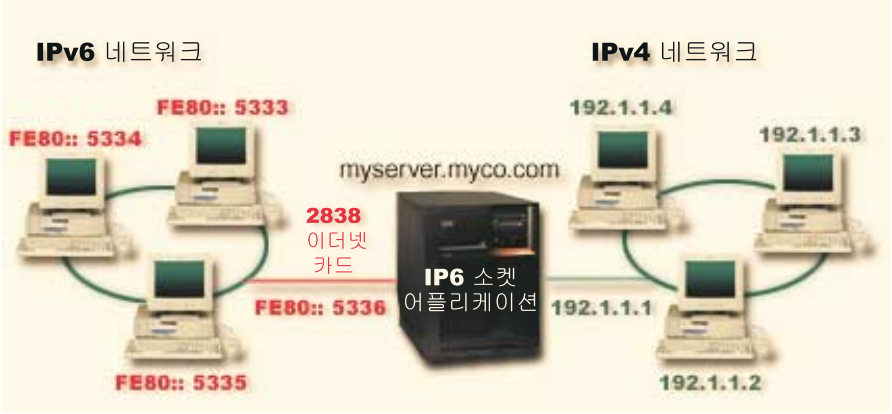
필수 단계

이런 목적에 맞는 어플리케이션을 개발하기 전에 다음 타스크를 완료하십시오.

1. QSYSINC 라이브러리를 설치하십시오. 이 라이브러리는 소켓 어플리케이션 컴파일시 필요한 필수 헤더 파일을 제공합니다.
2. C 컴파일러 사용권 프로그램(5722-CX2)을 설치하십시오.
3. 2838 이더넷 카드를 설치하고 구성하십시오. 이더넷 옵션에 대한 정보는 Information Center의 이더넷 주제를 참조하십시오.
4. TCP/IP 설정 및 IPv6 네트워크

시나리오 세부사항

다음 그래픽은 IPv6 및 IPv4 클라이언트로부터 요구를 처리하기 위해 어플리케이션을 작성하게 되는 IPv6 네트워크를 설명합니다. iSeries에는 이런 클라이언트로부터 요구를 청취하고 처리하는 프로그램이 포함되어 있습니다. 네트워크는 두 개의 별도 정의역으로 구성되는데 하나는 IPv4 클라이언트를 배타적으로 포함하고 있고 다른 정의역은 IPv6 클라이언트만 들어 있는 리모트 네트워크입니다. iSeries의 정의역명은 myserver.myco.com 입니다. 서버 어플리케이션은 AF_INET6 주소 패밀리를 사용하여 **bind()** 기능 호출에 지정된 in6addr_any로 이런 수신 요구를 처리합니다.



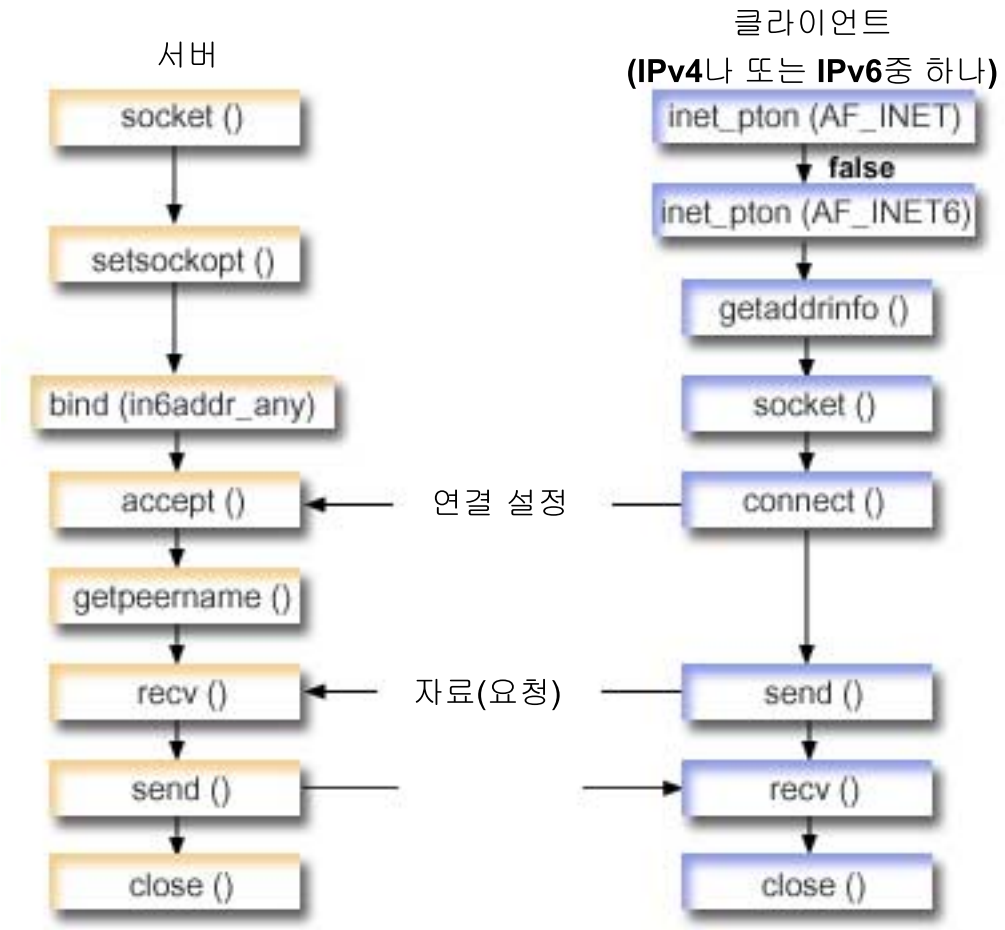
이 시나리오에서 사용되는 다음 프로그램 예를 참조하십시오.

- 예: IPv6 및 IPv4 클라이언트 모두로부터 연결 허용
- 예: IPv4 또는 IPv6 클라이언트

예: IPv6 및 IPv4 클라이언트 모두로부터 연결 허용

IPv4(AF_INET 주소 패밀리를 사용하는 소켓 어플리케이션) 및 IPv6(AF_INET6 주소 패밀리를 사용하는 어플리케이션) 모두로부터 요구를 허용하는 서버/클라이언트 모델을 작성하려면 이 샘플 프로그램을 사용하십시오. 현재는 사용자의 소켓 어플리케이션이 TCP 및 UDP 프로토콜을 허용하는 AF_INET 주소 패밀리만 사용할 수 있지만 IPv6 주소 사용이 증대하면 변경될 수 있습니다. 이 샘플 프로그램을 사용하여 두 주소 패밀리를 모두 수용하는 사용자 고유의 어플리케이션을 작성할 수 있습니다.

이 그래픽은 이 프로그램 예가 작동하는 방법을 나타낸 것입니다.



이벤트의 소켓 흐름: IPv4 및 IPv6 클라이언트 모두로부터 요구를 허용하는 서버 어플리케이션

이 흐름에서는 각각의 기능 호출과 IPv4 및 IPv6 클라이언트 모두로부터 요구를 허용하는 소켓 어플리케이션 내에서 기능 호출이 수행하는 일을 설명합니다.

1. **socket()** API는 종료점을 작성하는 소켓 설명자를 지정합니다. IPv6을 지원하는 AF_INET6 주소 패밀리, TCP 전송(SOCK_STREAM)이 이 소켓에 사용될 것임도 지정합니다.
2. **setsockopt()** 함수는 필수 대기 시간이 만기되기 전에 서버가 재시작될 때 어플리케이션이 로컬 주소를 재사용할 수 있도록 합니다.
3. **bind()** 함수는 소켓에 대해 고유한 이름을 제공합니다. 이 예에서 프로그래머는 주소를 in6addr_any로 설정하여 디폴트로 3005 포트를 지정하는 IPv4 또는 IPv6 클라이언트에서 연결을 설정할 수 있도록 합니다(즉, IPv4 및 IPv6 포트 공간 모두에 대해 바인드가 수행됩니다).

   주: 서버가 IPv6 클라이언트만 처리해야 할 경우에는 IPV6_ONLY 소켓 옵션을 사용할 수 있습니다.
4. **listen()** 함수는 서버가 수신 클라이언트 연결을 허용할 수 있도록 합니다. 이 예에서 프로그래머는 백로그를 10으로 설정하여 시스템이 수신 요구 거부를 시작하기 전에 10개의 연결 요구를 대기행렬에 넣을 수 있도록 합니다.

5. 서버에서는 수신 연결 요구를 승인하기 위해 **accept()** 함수를 사용합니다. **accept()** 호출은 IPv4 또는 IPv6 클라이언트로부터 수신 연결이 도착하길 기다리며 무기한 블록화됩니다.
6. **getpeername()** 함수는 클라이언트의 주소를 어플리케이션에 리턴합니다. 클라이언트가 IPv4 클라이언트인 경우 주소는 IPv4-맵핑 IPv6 주소로 표시됩니다.
7. **recv()** 함수는 클라이언트로부터 250바이트의 자료를 수신합니다. 이 예에서 프로그래머는 클라이언트가 250바이트의 자료를 송신한다는 것을 알고 있습니다. 이 사실을 알고 있기 때문에 SO_RCVLOWAT 소켓 옵션을 사용하여 250바이트의 자료가 모두 도착할 때까지 **recv()**가 깨어나지 않도록 지정합니다.
8. **send()** 함수는 클라이언트로 자료를 다시 에코우합니다.
9. **close()** 함수는 열린 소켓 설명자를 닫습니다.

이벤트의 소켓 흐름: IPv4 또는 IPv6 클라이언트로부터의 요구

주: 이 클라이언트 예는 IPv4 또는 IPv6 노드에 대해 요구를 허용할 다른 서버 어플리케이션 설계와 함께 사용할 수 있습니다. 예: 연결 지향 설계에 설명된 서버와 마찬가지로 다른 서버 설계를 이 클라이언트 예와 함께 사용할 수 있습니다.

1. **inet_pton()** 호출은 텍스트 주소 양식을 2진 양식으로 변환합니다. 이 예에서는 이런 호출이 두 개 발행됩니다. 첫 번째 호출은 서버가 유효한 AF_INET 주소인지를 판별합니다. 두 번째 호출은 **inet_pton()** 호출은 서버에 AF_INET6 주소가 있는지 여부를 판별합니다. 주소가 숫자인 경우 **getaddrinfo()**가 이름 분석을 수행하지 않도록 하려고 합니다. 그렇지 않으면 **getaddrinfo()** 호출이 발행될 때 분석되어야 하는 호스트명이 제공되었습니다.
2. **getaddrinfo()** 호출은 후속적인 **socket()** 및 **connect()** 호출에 필요한 주소 정보를 검색합니다.
3. **socket()** 함수는 종료점을 표시하는 소켓 설명자를 리턴합니다. 이 명령문은 **getaddrinfo()**에서 리턴된 정보를 사용하여 주소 패밀리, 소켓 유형 및 프로토콜도 식별합니다.
4. **connect()** 함수는 서버가 IPv4인지 IPv6인지에 관계없이 서버와의 연결을 설정합니다.
5. **send()** 함수는 서버로 자료 요구를 송신합니다.
6. **recv()** 함수는 서버 어플리케이션으로부터 자료를 수신합니다.
7. **close()** 함수는 열린 소켓 설명자를 닫습니다.

다음 샘플 코드는 이 시나리오의 서버 어플리케이션을 나타낸 것입니다. 클라이언트 어플리케이션은 예: IPv4 또는 IPv6 클라이언트를 참조하십시오. 이 코드를 사용하는 것에 대한 정보는 코드 면책사항 정보을 참조하십시오.

```
/*********************************************************************/
/* Header files needed for this sample program                       */
/*********************************************************************/
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

/*********************************************************************/
/* Constants used by this program                                    */
```

```
/*********************************************************************/
#define SERVER_PORT     3005
#define BUFFER_LENGTH    250
#define FALSE            0

void main()
{
   /*******************************************************************/
  /* Variable and structure definitions. */
   /*******************************************************************/
   int sd=-1, sdconn=-1;
   int rc, on=1, rcdsize=BUFFER_LENGTH;
   char buffer[BUFFER_LENGTH];
   struct sockaddr_in6 serveraddr, clientaddr;
   int addrlen=sizeof(clientaddr);
   char str[INET6_ADDRSTRLEN];

   /*******************************************************************/
   /* A do/while(FALSE) loop is used to make error cleanup easier.  The   */
   /* close() of each of the socket descriptors is only done once at the  */
   /* very end of the program.                                            */
   /*******************************************************************/
   do
   {

      /****************************************************************/
      /* The socket() function returns a socket descriptor representing   */
      /* an endpoint.  Get a socket for address family AF_INET6 to        */
      /* prepare to accept incoming connections on.                       */
      /****************************************************************/
      if ((sd = socket(AF_INET6, SOCK_STREAM, 0)) < 0)
      {
    perror("socket() failed");
         break;
      }

      /****************************************************************/
      /* The setsockopt() function is used to allow the local address to  */
      /* be reused when the server is restarted before the required wait  */
      /* time expires.                                                    */
      /****************************************************************/
     if (setsockopt(sd, SOL_SOCKET, SO_REUSEADDR,
                    (char *)&on,sizeof(on)) < 0)
      {
         perror("setsockopt(SO_REUSEADDR) failed");
         break;
      }

      /****************************************************************/
      /* After the socket descriptor is created, a bind() function gets a */
      /* unique name for the socket.  In this example, the user sets the  */
      /* address to in6addr_any, which (by default) allows connections to */
      /* be established from any IPv4 or IPv6 client that specifies port  */
      /* 3005. (i.e. the bind is done to both the IPv4 and IPv6 TCP/IP    */
      /* stacks).  This behavior can be modified using the IPPROTO_IPV6   */
      /* level socket option IPV6_V6ONLY if desired.                      */
      /****************************************************************/
      memset(&serveraddr, 0, sizeof(serveraddr));
```

```
      serveraddr.sin6_family = AF_INET6;
      serveraddr.sin6_port   = htons(SERVER_PORT);
      /*********************************************************************/
      /* Note: applications use in6addr_any similarly to the way they use */
      /* INADDR_ANY in IPv4.  A symbolic constant IN6ADDR_ANY_INIT also   */
      /* exists but can only be used to initialize an in6_addr structure  */
      /* at declaration time (not during an assignment).                  */
      /*********************************************************************/
      serveraddr.sin6_addr   = in6addr_any;
      /*********************************************************************/
      /* Note: the remaining fields in the sockaddr_in6 are currently not */
      /* supported and should be set to 0 to ensure upward compatibility. */
      /*********************************************************************/

      if (bind(sd,
               (struct sockaddr *)&serveraddr,
                sizeof(serveraddr)) < 0)
      {
     perror("bind() failed");
         break;
      }

      /*********************************************************************/
      /* The listen() function allows the server to accept incoming       */
      /* client connections.  In this example, the backlog is set to 10.  */
      /* This means that the system will queue 10 incoming connection     */
      /* requests before the system starts rejecting the incoming         */
      /* requests.                                                        */
      /*********************************************************************/
      if (listen(sd, 10) < 0)
      {
     perror("listen() failed");
         break;
      }

   printf("Ready for client connect().\n");

      /*********************************************************************/
      /* The server uses the accept() function to accept an incoming      */
      /* connection request.  The accept() call will block indefinitely   */
      /* waiting for the incoming connection to arrive from an IPv4 or    */
      /* IPv6 client.                                                     */
      /*********************************************************************/
      if ((sdconn = accept(sd, NULL, NULL)) < 0)
      {
      perror("accept() failed");
         break;
      }
    else
      {
         /*****************************************************************/
         /* Display the client address.  Note that if the client is       */
         /* an IPv4 client, the address will be shown as an IPv4 Mapped   */
         /* IPv6 address.                                                 */
         /*****************************************************************/
         getpeername(sdconn, (struct sockaddr *)&clientaddr, &addrlen);
         if(inet_ntop(AF_INET6, &clientaddr.sin6_addr, str, sizeof(str))) {
            printf("Client address is %s\n", str);
```

```
|              printf("Client port is %d\n", ntohs(clientaddr.sin6_port));
|           }
|        }
|
|        /*******************************************************************/
|        /* In this example we know that the client will send 250 bytes of   */
|        /* data over.  Knowing this, we can use the SO_RCVLOWAT socket      */
|        /* option and specify that we don't want our recv() to wake up      */
|        /* until all 250 bytes of data have arrived.                        */
|        /*******************************************************************/
|        if (setsockopt(sdconn, SOL_SOCKET, SO_RCVLOWAT,
|                       (char *)&rcdsize,sizeof(rcdsize)) < 0)
|        {
|           perror("setsockopt(SO_RCVLOWAT) failed");
|           break;
|        }
|
|        /*******************************************************************/
|        /* Receive that 250 bytes of data from the client                   */
|        /*******************************************************************/
|        rc = recv(sdconn, buffer, sizeof(buffer), 0);
|        if (rc < 0)
|        {
|        perror("recv() failed");
|           break;
|        }
|
|        printf("%d bytes of data were received\n", rc);
|        if (rc == 0 ||
|            rc < sizeof(buffer))
|        {
|           printf("The client closed the connection before all of the\n");
|           printf("data was sent\n");
|           break;
|        }
|
|        /*******************************************************************/
|     /* Echo the data back to the client              */
|        /*******************************************************************/
|        rc = send(sdconn, buffer, sizeof(buffer), 0);
|        if (rc < 0)
|        {
|        perror("send() failed");
|           break;
|        }
|
|        /*******************************************************************/
|        /* Program complete                                                 */
|        /*******************************************************************/
|
|     } while (FALSE);
|
|     /*******************************************************************/
|     /* Close down any open socket descriptors                          */
|     /*******************************************************************/
|     if (sd != -1)
```

```
            close(sd);
   if (sdconn != -1)
      close(sdconn);
}
```

예: IPv4 또는 IPv6 클라이언트

샘플 프로그램을 IPv4 또는 IPv6 클라이언트로부터 요구를 허용하는 서버 어플리케이션과 함께 사용할 수 있습니다.

```
/*********************************************************************/
/* This is an IPv4 or IPv6 client.                                   */
/*********************************************************************/

/*********************************************************************/
/* Header files needed for this sample program                       */
/*********************************************************************/
#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>

/*********************************************************************/
/* Constants used by this program                                    */
/*********************************************************************/
#define BUFFER_LENGTH    250
#define FALSE            0
#define SERVER_NAME     "ServerHostName"

/* Pass in 1 parameter which is either the */
/* address or host name of the server, or  */
/* set the server name in the #define      */
/* SERVER_NAME.                            */
void main(int argc, char *argv[])
{
   /*******************************************************************/
  /* Variable and structure definitions. */
   /*******************************************************************/
   int    sd=-1, rc, bytesReceived=0;
   char buffer[BUFFER_LENGTH];
   char   server[NETDB_MAX_HOST_NAME_LENGTH];
   char   servport[] = "3005";
   struct in6_addr serveraddr;
   struct addrinfo hints, *res=NULL;

   /*******************************************************************/
   /* A do/while(FALSE) loop is used to make error cleanup easier.  The   */
   /* close() of the socket descriptor is only done once at the very end  */
   /* of the program along with the free of the list of addresses.        */
 /*********************************************************************/
   do
   {
      /****************************************************************/
      /* If an argument was passed in, use this as the server, otherwise  */
```

```
      /* use the #define that is located at the top of this program.      */
      /*********************************************************************/
   if (argc > 1)
         strcpy(server, argv[1]);
    else
         strcpy(server, SERVER_NAME);

      memset(&hints, 0x00, sizeof(hints));
      hints.ai_flags    = AI_NUMERICSERV;
      hints.ai_family   = AF_UNSPEC;
      hints.ai_socktype = SOCK_STREAM;
      /*********************************************************************/
      /* Check if we were provided the address of the server using        */
      /* inet_pton() to convert the text form of the address to binary    */
      /* form. If it is numeric then we want to prevent getaddrinfo()     */
      /* from doing any name resolution.                                  */
      /*********************************************************************/
      rc = inet_pton(AF_INET, server, &serveraddr);
      if (rc == 1)    /* valid IPv4 text address? */
      {
         hints.ai_family = AF_INET;
            hints.ai_flags |= AI_NUMERICHOST;
      }
    else
      {
         rc = inet_pton(AF_INET6, server, &serveraddr);
         if (rc == 1) /* valid IPv6 text address? */
         {

            hints.ai_family = AF_INET6;
            hints.ai_flags |= AI_NUMERICHOST;
         }
      }
      /*********************************************************************/
      /* Get the address information for the server using getaddrinfo().  */
      /*********************************************************************/
      rc = getaddrinfo(server, servport, &hints, &res);
   if (rc != 0)
      {
         printf("Host not found --> %s\n", gai_strerror(rc));
         if (rc == EAI_SYSTEM)
            perror("getaddrinfo() failed");
         break;
      }

      /*********************************************************************/
      /* The socket() function returns a socket descriptor representing   */
      /* an endpoint.  The statement also identifies the address family,  */
      /* socket type, and protocol using the information returned from    */
      /* getaddrinfo().                                                   */
      /*********************************************************************/
      sd = socket(res->ai_family, res->ai_socktype, res->ai_protocol);
     if (sd < 0)
      {
     perror("socket() failed");
         break;
      }
      /*********************************************************************/
```

```
    /* Use the connect() function to establish a connection to the      */
                /* server.                                              */
    /*********************************************************************/
    rc = connect(sd, res->ai_addr, res->ai_addrlen);
    if (rc < 0)
    {
       /*********************************************************************/
       /* Note: the res is a linked list of addresses found for server. */
       /* If the connect() fails to the first one, subsequent addresses */
       /* (if any) in the list could be tried if desired.               */
       /*********************************************************************/
  perror("connect() failed");
       break;
    }

    /*********************************************************************/
    /* Send 250 bytes of a's to the server                               */
    /*********************************************************************/
    memset(buffer, 'a', sizeof(buffer));
    rc = send(sd, buffer, sizeof(buffer), 0);
    if (rc < 0)
    {
    perror("send() failed");
       break;
    }

    /*********************************************************************/
    /* In this example we know that the server is going to respond with */
    /* the same 250 bytes that we just sent.  Since we know that 250    */
    /* bytes are going to be sent back to us, we could use the          */
    /* SO_RCVLOWAT socket option and then issue a single recv() and     */
    /* retrieve all of the data.                                        */
    /*                                                                  */
    /* The use of SO_RCVLOWAT is already illustrated in the server      */
    /* side of this example, so we will do something different here.    */
    /* The 250 bytes of the data may arrive in separate packets,        */
    /* therefore we will issue recv() over and over again until all     */
    /* 250 bytes have arrived.                                          */
    /*********************************************************************/
    while (bytesReceived < BUFFER_LENGTH)
    {
       rc = recv(sd, & buffer[bytesReceived],
                 BUFFER_LENGTH - bytesReceived, 0);
    if (rc < 0)
       {
    perror("recv() failed");
       break;
       }
      else if (rc == 0)
       {
          printf("The server closed the connection\n");
       break;
       }

       /*********************************************************************/
       /* Increment the number of bytes that have been received so far  */
       /*********************************************************************/
       bytesReceived += rc;
```

```
        }

    } while (FALSE);

    /*********************************************************************/
    /* Close down any open socket descriptors                            */
    /*********************************************************************/
    if (sd != -1)
             close(sd);
    /*********************************************************************/
    /* Free any results returned from getaddrinfo                        */
    /*********************************************************************/
    if (res != NULL)
       freeaddrinfo(res);
 }
```

제 9 장 소켓 어플리케이션 설계 권장사항

소켓 어플리케이션에 대해 작업하기 전에, 기능적 요구사항, 목표 및 소켓 어플리케이션의 요구에 액세스하십시오. 어플리케이션의 성능 요구사항 및 시스템 자원 효과도 고려하십시오. 다음 권장사항 리스트는 소켓 어플리케이션에 대한 일부 문제를 제시하며 소켓 사용 및 소켓 어플리케이션 설계에 대한 더 나은 방법을 알려주는 데 도움을 줍니다.

표 19. 소켓 설계 어플리케이션

권장사항	이유	최상의 사용
비동기 I/O 사용	스레드 서버 모델에 사용된 I/O는 보다 일반적인 select() 모델보다 더 좋습니다. 비동기 I/O 사용 장점에 대한 자세한 정보는 비동기 I/O를 참조하십시오. 비동기 I/O API를 사용하는 샘플 프로그램은 예: 비동기 I/O 사용을 참조하십시오.	다수의 동시 클라이언트를 처리 중인 소켓 서버 어플리케이션.
비동기 I/O 사용시, 처리할 클라이언트 수의 경우 최적의 수로 프로세스에서 스레드 수를 조정하십시오.	너무 적은 스레드가 정의되어 있으면 일부 클라이언트에서는 처리 전에 시간 종료될 수 있습니다. 너무 많은 스레드가 정의되어 있으면 일부 시스템 자원이 효율적으로 사용되지 않습니다. **주:** 너무 적은 스레드보다는 너무 많은 스레드를 갖는 것이 더 좋습니다.	비동기 I/O를 사용한 소켓 어플리케이션
비동기 I/O의 모든 시작 조작시 postflag 사용을 피하도록 소켓 어플리케이션을 설계하십시오.	조작이 이미 동기적으로 만족된 경우 완료 포트에 대한 전송의 성능 오버헤드를 피합니다.	비동기 I/O를 사용한 소켓 어플리케이션
read() 및 **write()**상에서 **send()** 및 **recv()**를 사용하십시오.	**send()** 및 **recv()** API는 **read()** 및 **write()** 상에 작은 성능 및 서비스 능력 향상을 제공합니다.	이를 알고 있는 모든 소켓 프로그램에서는 파일 설명자가 아니라 소켓 설명자를 사용합니다.
모든 자료가 도달할 때까지 수신 조작에서 루프를 피하도록, 수신 최저 (SO_RCVLOWAT) 소켓 옵션을 사용하십시오.	어플리케이션에서는 블록화된 수신 조작을 만족하기 전에 소켓에서 최소 자료량이 수신되도록 기다릴 수 있습니다.	자료를 수신하는 모든 소켓 어플리케이션
모든 자료가 도착할 때까지 수신 조작에서 루프를 피하도록 하려면 MSG_WAITALL 플래그를 사용하십시오.	어플리케이션이 블록화된 수신 조작을 만족하기 전에 수신 조작에 제공된 전체 버퍼가 수신될 때까지 기다릴 수 있도록 합니다.	자료를 수신하고 도착할 자료 양을 미리 알고 있는 소켓 어플리케이션.
givedescriptor() 및 **takedescriptor()**상에서 **sendmsg()** 및 **recvmsg()**를 사용하십시오.	장점을 보려면 프로세스간의 설명자 전달--sendmsg() 및 recvmsg()를 참조하십시오. **sendmsg()** 및 **recvmsg()**를 사용하는 샘플 프로그램은 예: 프로세스간의 설명자 전달을 참조하십시오.	프로세스간의 소켓 또는 파일 설명자를 전달하는 모든 소켓 어플리케이션.

표 19. 소켓 설계 어플리케이션 (계속)

select() 사용시, read, write 또는 exception 세트에서 다수의 설명자를 피하도록 하십시오. **주:** **select()** 처리에 사용되고 있는 다수의 설명자가 있으면 위의 비동기 I/O 권장사항을 참조하십시오.	read, write 또는 exception 세트에 다수의 설명자가 있으면 **select()**가 호출될 때마다 상당한 불필요한 작업이 발생합니다. 일단 **select()**에 만족하면 실제 소켓 함수는 여전히 수행되어야 합니다. 즉, read, write 또는 accept이 여전히 수행되어야 합니다. 비동기 I/O API는 실제 I/O 조작과 소켓에 어떤 상황이 발생했다는 통지를 결합시킵니다.	**select()**에 활동 중인 큰 수의 설명자(>50)가 있는 어플리케이션.
select()를 다시 발행해야 할 때마다 해당 세트를 리빌드하지 않도록 **select()**를 사용하기 전에 read, write 및 exception 세트의 사본을 저장하십시오.	이렇게 하면 **select()**를 발행하고자 할 때마다 read, write 또는 exception 세트의 리빌드 오버헤드가 저장됩니다. **select()**를 사용하는 샘플 프로그램은 예: 비블록화 I/O 및 select()를 참조하십시오.	read, write 또는 exception 처리에 작동할 수 있는 다수의 소켓으로 **select()**를 사용 중인 모든 소켓 어플리케이션.
타이머로 **select()**를 사용하지 마십시오. 대신에 **sleep()**을 사용하십시오. **주:** **sleep()** 타이머의 입도가 적합하지 않으면 타이머로 **select()**를 사용해야 할 수 있습니다. 이 경우 최대 설명자를 0으로 설정하고 read, write 및 exception 세트를 널(null)로 설정하십시오.	타이머 응답이 나아지면 시스템 오버헤드는 줄어듭니다.	타이머로만 **select()**를 사용 중인 모든 소켓 어플리케이션.
소켓 어플리케이션에서 DosSetRelMaxFH()를 사용하여 프로세스당 허용되는 최대 파일 수 및 소켓 설명자 수가 증가하고 이와 동일한 어플리케이션에서 **select()**를 사용 중이면 이러한 새로운 최대값이 **select()** 처리에 사용된 read, write 및 exception 세트의 크기에 주는 영향에 유의하십시오.	FD_SETSIZE에 지정된 대로 read, write 또는 exception 세트의 범위를 초과하여 설명자를 할당하면 기억장치에 겹쳐쓰거나 기억장치가 손상될 수 있습니다. 세트 크기는 **select()** API에 지정된 최대 설명자 값과 프로세스에 설정된 최대 설명자 수가 무엇이든 지간에 적어도 처리하기에는 충분해야 합니다.	DosSetRelMaxFH() 및 **select()**를 사용하는 모든 어플리케이션 또는 프로세스.
read 또는 write 세트의 모든 소켓 설명자를 비블록화로 설정하십시오. 설명자가 read 또는 write에 작동할 수 있게 되면 EWOULDBLOCK이 리턴될 때까지 모든 자료를 루프 및 사용하거나 송신하십시오. **select()**를 사용하는 샘플 프로그램은 예: 비블록화 I/O 및 select()를 참조하십시오.	이로써 사용자는 자료를 여전히 처리할 수 있거나 설명자에서 읽을 수 있을 때 **select()** 호출의 수를 최소화할 수 있습니다.	**select()**를 사용 중인 모든 소켓 어플리케이션.
select() 처리에 사용해야 하는 세트만 지정하십시오.	대부분의 어플리케이션에서는 exception 세트나 write 세트를 지정해서는 안됩니다.	**select()**를 사용 중인 모든 소켓 어플리케이션.
SSL API 대신에 GSKiet API를 사용하십시오.	글로벌 보안 툴킷(GSKit)과 OS/400 SSL_ API 모두 보안 AF_INET 또는 AF_INET6, SOCK_STREAM 소켓 어플리케이션을 개발할 수 있도록 합니다. GSKit API는 IBM@server 플랫폼 전반에 걸쳐 지원되기 때문에 보안 어플리케이션에 선호되는 API 세트입니다. SSL_ API는 OS/400 시스템에만 고유합니다.	SSL/TLS 처리에 작동할 수 있어야 하는 모든 소켓 어플리케이션.

표 19. 소켓 설계 어플리케이션 (계속)

신호 사용을 피하십시오.	신호의 성능 오버헤드(iSeries만이 아니라 모든 플랫폼에서)는 비용이 많이 듭니다. 비동기 I/O나 **select()** API를 사용하기 위해 소켓 어플리케이션을 설계하는 것이 더 좋습니다.	해당 소켓 어플리케이션에서 신호 사용을 고려하는 모든 프로그래머.
사용할 수 있는 경우 **inet_ntop()**, **inet_pton()**, **getaddrinfo()** 및 **getnameinfo()**와 같은 독립 루틴을 사용하십시오.	IPv6을 지원할 준비가 되지 않았더라도 수월한 마이그레이션을 준비하려면 이런 API를 사용(**inet_ntoa()**, **inet_addr()**, **gethostbyname()** 및**gethostbyaddr**() 대신)하십시오.	네트워크 루틴을 사용하는 AF_INET 또는 AF_INET6 어플리케이션.
주소 패밀리 주소에 대한 기억장치를 선언하려면 sockaddr_storage를 사용하십시오.	복수 주소 패밀리 및 플랫폼간에 이식가능 코드 작성을 단순화합니다. 최대 주소 패밀리를 보유할 수 있을 만큼 대형 기억장치를 선언하고 올바른 경계 정렬을 보장합니다.	주소를 저장하는 모든 소켓 어플리케이션

제 10 장 예: 소켓 어플리케이션 설계

다음 예에서는 보다 확장된 소켓 개념을 보여주는 다수의 샘플 프로그램이 제공됩니다. 이 샘플 프로그램을 사용하여 유사한 타스크를 완료하는 사용자 고유의 어플리케이션을 작성할 수 있습니다. 이 예에는 이런 각각의 어플리케이션에서 이벤트의 흐름을 설명하는 호출 리스팅 및 그래픽이 있습니다. Xsocket 툴을 사용하여 이런 프로그램에서 이런 API 중 일부를 대화식으로 시도하거나 특정 환경에 맞게 특정 변경사항을 작성할 수 있습니다.

- 예: 연결 지향 설계
- 예: 소스 연결 설정
- 예: 스레드세이프 네트워크 루틴에 gethostbyaddr_r() 사용
- 예: 비블록화 I/O 및 select()
- 예: 블록화 소켓 API에 신호 사용
- AF_INET 주소 패밀리에 멀티캐스팅 사용
- 예: DNS 조회 및 갱신
- 예: send_file() 및 accept_and_recv() API를 사용하여 파일 자료 전송

예: 연결 지향 설계

iSeries에서 연결 지향 소켓 서버를 설계할 수 있는 방법이 많이 있습니다. 다음 프로그램 예를 사용하여 사용자 고유의 연결 지향 설계를 작성할 수 있습니다. 추가로 소켓 서버 설계를 할 수 있긴 하지만 아래 예에 제공된 설계가 가장 일반적입니다.

반복 서버

반복 서버 예에서는 하나의 서버 작업이 모든 수신 연결 및 클라이언트와의 자료 흐름을 모두 처리합니다. **accept()** API가 완료될 때 서버는 전체 트랜잭션을 처리합니다. 이것은 개발하기 가장 쉬운 서버이긴 하지만 몇 가지 문제점이 있습니다. 서버가 지정 클라이언트로부터 요구를 처리하는 동안 추가 클라이언트가 서버에 연결을 시도할 수 있습니다. 이런 요구가 **listen()** 백로그를 채우고 결국 그 중 일부는 거부됩니다.

나머지 예는 모두 동시 서버 설계입니다. 이 설계에서 시스템은 다중 작업 및 스레드를 사용하여 수신 연결 요구를 처리합니다. 동시 서버의 경우에는 대개 서버에 동시에 연결하는 클라이언트가 여러 개 있습니다.

네트워크에 동시 클라이언트가 여러 개 있을 경우에는 비동기 I/O 소켓 API를 사용할 것을 권장합니다. 이런 API는 복수 동시 클라이언트를 갖는 네트워크에서 최상의 성능을 제공합니다. 비동기 I/O에서는 이런 API가 수행하는 일과 작동 방법을 설명합니다. 이런 API를 사용하는 프로그램 예는 예: 비동기 I/O 사용을 참조하십시오.

spawn() 서버 및 spawn() 작업자

spawn() 서버 및 spawn() 작업자 예에서는 spawn() API를 사용하여 각각의 수신 요구를 처리하기 위한 새로운 작업을 작성합니다. spawn()이 완료되면 서버가 **accept()** API에서 다음 수신 연결이 수신되길 기다릴 수 있습니다.

이 서버 설계의 유일한 문제점은 연결이 수신될 때마다 새로운 작업을 작성하는 성능 오버헤드입니다. 사전시작된 작업을 사용하면 spawn() 서버 예의 성능 오버헤드를 피할 수 있습니다. 연결이 수신될 때마다 새로운 작업을 작성하는 대신 이미 사용 중인 작업에 수신 연결이 제공됩니다. 이 주제의 나머지 예는 모두 사전시작된 작업을 사용합니다.

sendmsg() server 및 recvmsg() worker

sendmsg() 서버 및 recvmsg() 작업자 예는 수신 연결을 작업자(클라이언트) 작업에 전달합니다. 서버는 서버 작업이 처음 시작될 때 작업자 작업을 모두 사전시작했습니다.

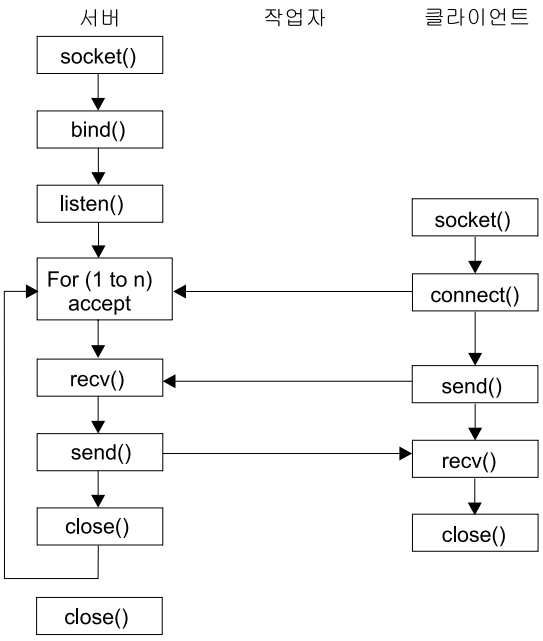
복수 accept() 서버 및 복수 accept() 작업자

이전 예에서는 서버가 수신 연결 요구를 수신한 후까지 작업자 작업이 관련되지 않았습니다. 시스템의 복수 accept() 서버 및 복수 accept() 작업자 예는 각각의 작업자 작업을 반복 서버로 전달합니다. 서버 작업은 여전히 **socket()**, **bind()** 및 **listen()** API를 호출합니다. **listen()** 호출이 완료되면 서버가 작업자 작업을 각각 작성하고 각 작업자 작업 중 하나에 청취 소켓을 제공합니다. 그런 다음 작업자 작업이 모두 **accept()** API를 호출합니다. 클라이언트가 서버에 연결하려고 시도할 때 하나의 **accept()** 호출만 완료되며 해당 작업자가 연결을 처리합니다.

이런 예는 모두 클라이언트 연결에 기본 설계를 사용합니다. 자세한 내용은 예: 총칭 클라이언트를 참조하십시오.

예: 반복 서버 프로그램 작성

수신 연결을 모두 처리하는 단일 서버 작업을 작성하려면 이 예를 사용하십시오. **accept()** API가 완료될 때 서버는 전체 트랜잭션을 처리합니다. 그림은 시스템이 서버 설계를 사용했을 때 서버와 클라이언트 작업이 상호작용하는 방법을 보여줍니다. 이 예에 사용할 수 있는 공통 클라이언트 작업의 코드가 들어 있는 예를 보려면, 예: 총칭 클라이언트를 참조하십시오.



소켓 이벤트의 흐름: 반복 서버

다음 소켓 호출 순서는 그래픽에 대한 설명을 제공합니다. 서버와 클라이언트 어플리케이션간 관계도 설명합니다. 각각의 흐름 세트에는 특정 API에서의 사용법 노트로의 링크가 포함되어 있습니다. 특정 API 사용에 대한 추가 정보가 필요하면 이런 링크를 사용할 수 있습니다. 이 흐름 중 클라이언트 부분에 대한 설명은 예: 총칭 클라이언트를 참조하십시오. 다음 순서는 이 샘플 프로그램 및 반복 서버 어플리케이션에 대한 기능 호출을 나타냅니다.

1. **socket()** 함수는 종료점을 표시하는 소켓 설명자를 리턴합니다. 이 명령문은 TCP 전송(SOCK_STREAM)을 사용하는 INET(인터넷 프로토콜) 주소 패밀리가 이 소켓에 사용된다는 것도 식별합니다.
2. 소켓 설명자가 작성된 후에 **bind()** 함수는 소켓에 고유한 이름을 제공합니다.
3. **listen()**은 서버가 수신 클라이언트 연결을 허용할 수 있도록 합니다.
4. 서버에서는 수신 연결 요구를 승인하기 위해 **accept()** 함수를 사용합니다. **accept()** 호출은 수신 연결이 도착하길 기다리며 무기한 블록화됩니다.
5. **recv()** 함수는 클라이언트 어플리케이션으로부터 자료를 수신합니다.
6. **send()** 함수는 클라이언트로 자료를 다시 에코우합니다.
7. **close()** 함수는 열린 소켓 설명자를 닫습니다.

```
/*****************************************************************************/
/* Application creates an iterative server design                            */
/*****************************************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <sys/socket.h>
#include <netinet/in.h>

#define SERVER_PORT 12345

main (int argc, char *argv[])
```

```
{
   int    i, len, num, rc, on = 1;
   int    listen_sd, accept_sd;
   char   buffer[80];
   struct sockaddr_in   addr;

   /*************************************************/
   /* If an argument was specified, use it to       */
   /* control the number of incoming connections    */
   /*************************************************/
   if (argc >= 2)
      num = atoi(argv[1]);
   else
      num = 1;

   /*************************************************/
   /* Create an AF_INET stream socket to receive    */
   /* incoming connections on                       */
   /*************************************************/
   listen_sd = socket(AF_INET, SOCK_STREAM, 0);
   if (listen_sd < 0)
   {
    perror("socket() failed");
          exit(-1);
   }

   /*************************************************/
  /* Allow socket descriptor to be reuseable */
   /*************************************************/
   rc = setsockopt(listen_sd,
                   SOL_SOCKET,  SO_REUSEADDR,
                   (char *)&on, sizeof(on));
      if (rc < 0)
   {
    perror("setsockopt() failed");
            close(listen_sd);
          exit(-1);
   }

   /*************************************************/
   /* Bind the socket                               */
   /*************************************************/
   memset(&addr, 0, sizeof(addr));
   addr.sin_family = AF_INET;
      addr.sin_addr.s_addr = htonl(INADDR_ANY);
   addr.sin_port   = htons(SERVER_PORT);
   rc = bind(listen_sd,
             (struct sockaddr *)&addr, sizeof(addr));
      if (rc < 0)
   {
    perror("bind() failed");
            close(listen_sd);
          exit(-1);
   }

   /*************************************************/
   /* Set the listen back log                       */
   /*************************************************/
```

```
    rc = listen(listen_sd, 5);
       if (rc < 0)
    {
     perror("listen() failed");
             close(listen_sd);
           exit(-1);
    }

    /*************************************************/
    /* Inform the user that the server is ready      */
    /*************************************************/
    printf("The server is ready\n");

    /*************************************************/
    /* Go through the loop once for each connection  */
    /*************************************************/
    for (i=0; i < num; i++)
    {
       /*********************************************/
    /* Wait for an incoming connection               */
       /*********************************************/
       printf("Interation: %d\n", i+1);
       printf("  waiting on accept()\n");
    accept_sd = accept(listen_sd, NULL, NULL);
    if (accept_sd < 0)
       {
       perror("accept() failed");
             close(listen_sd);
           exit(-1);
       }
       printf("  accept completed successfully\n");

       /*********************************************/
    /* Receive a message from the client             */
       /*********************************************/
       printf("  wait for client to send us a message\n");
    rc = recv(accept_sd, buffer, sizeof(buffer), 0);
    if (rc <= 0)
       {
       perror("recv() failed");
             close(listen_sd);
       close(accept_sd);
           exit(-1);
       }
    printf("<%s>\n", buffer);

       /*********************************************/
    /* Echo the data back to the client              */
       /*********************************************/
       printf("  echo it back\n");
    len = rc;
    rc = send(accept_sd, buffer, len, 0);
    if (rc <= 0)
       {
       perror("send() failed");
             close(listen_sd);
       close(accept_sd);
           exit(-1);
```

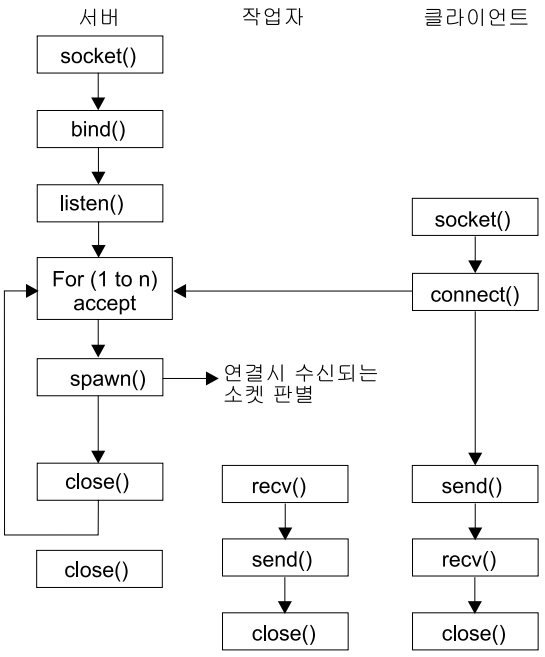
```
        }

        /*********************************************/
    /* Close down the incoming connection           */
        /*********************************************/
        close(accept_sd);
    }

    /*********************************************/
    /* Close down the listen socket                */
    /*********************************************/
            close(listen_sd);
}
```

예: spawn() API를 사용한 하위 프로세스 작성

이 예에서는 서버 프로그램이 **spawn()** API를 사용하여 상위 프로세스에서 소켓 설명자를 계승하는 하위 프로세스를 작성하는 방법을 보여줍니다. 서버 작업은 수신 연결을 기다린 후 **spawn()**을 호출하여 수신 연결을 처리하기 위한 하위 작업을 작성합니다. 하위 프로세스는 **spawn()** 함수를 사용하여 다음 속성을 계승합니다.

- 소켓 및 파일 설명자
- 신호 마스크
- 신호 조치 벡터
- 환경 변수

다음 그림은 **spawn()** 서버 설계가 사용될 때 서버, 작업자 및 클라이언트 작업이 상호작용하는 방법을 보여줍니다.



소켓 이벤트의 흐름: spawn()을 사용하여 요구를 허용하고 처리하는 서버

다음 소켓 호출 순서는 그래픽에 대한 설명을 제공합니다. 서버와 작업자 예 사이의 관계도 설명합니다. 각각의 흐름 세트에는 특정 API에서의 사용법 노트로의 링크가 포함되어 있습니다. 특정 API 사용에 대한 추가

정보가 필요하면 이런 링크를 사용할 수 있습니다. 예: spawn()을 사용하는 서버 작성에서는 **spawn()** 기능 호출로 하위 프로세스를 작성하기 위해 다음 소켓 호출을 사용합니다.

1. **socket()** 함수는 종료점을 표시하는 소켓 설명자를 리턴합니다. 이 명령문은 TCP 전송(SOCK_STREAM)을 사용하는 INET(인터넷 프로토콜) 주소 패밀리가 이 소켓에 사용된다는 것도 식별합니다.
2. 소켓 설명자가 작성된 후에 **bind()** 함수는 소켓에 고유한 이름을 제공합니다.
3. **listen()**은 서버가 수신 클라이언트 연결을 허용할 수 있도록 합니다.
4. 서버에서는 수신 연결 요구를 승인하기 위해 **accept()** 함수를 사용합니다. **accept()** 호출은 수신 연결이 도착하길 기다리며 무기한 블록화됩니다.
5. **spawn()** 함수는 작업자 작업에 대한 매개변수를 초기화하여 수신 요구를 처리합니다. 이 예에서 새로운 연결을 위한 소켓 설명자는 하위 프로그램에서 설명자 0에 대해 맵핑됩니다.
6. 이 예에서 첫 번째 **close()** 함수는 청취 소켓 설명자를 닫습니다. 두 번째 **close()** 호출은 허용된 소켓을 종료합니다.

이벤트의 소켓 흐름: spawn()으로 작성된 작업자 작업

예: 작업자 작업이 자료 버퍼를 수신할 수 있도록 허용에서는 다음 기능 호출 순서를 사용합니다.

1. 서버에서 **spawn()** 함수가 호출된 후에 **recv()** 함수는 수신 연결로부터 자료를 수신합니다.
2. **send()** 함수는 클라이언트로 자료를 다시 에코우합니다.
3. **close()** 함수는 파생된(spawned) 작업자 작업을 종료합니다.

예: spawn()을 사용하는 서버 작성

다음 예에서는 **spawn()** API를 사용하여 상위 프로세스에서 소켓 설명자를 상속하는 하위 프로세스를 작성하는 방법을 보여줍니다. 코드 예를 사용하는 것에 대한 정보는 코드 면책사항을 참조하십시오.

```
/*********************************************************************/
/* Application creates an child process using spawn().               */
/*********************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <spawn.h>

#define SERVER_PORT 12345

main (int argc, char *argv[])
{
   int    i, num, pid, rc, on = 1;
   int    listen_sd, accept_sd;
   int    spawn_fdmap[1];
   char  *spawn_argv[1];
   char  *spawn_envp[1];
   struct inheritance   inherit;
   struct sockaddr_in   addr;

   /*************************************************/
   /* If an argument was specified, use it to       */
```

```
    /* control the number of incoming connections    */
    /************************************************/
    if (argc >= 2)
       num = atoi(argv[1]);
    else
       num = 1;

    /************************************************/
    /* Create an AF_INET stream socket to receive   */
    /* incoming connections on                      */
    /************************************************/
    listen_sd = socket(AF_INET, SOCK_STREAM, 0);
    if (listen_sd < 0)
    {
     perror("socket() failed");
           exit(-1);
    }

    /************************************************/
   /* Allow socket descriptor to be reuseable */
    /************************************************/
    rc = setsockopt(listen_sd,
                    SOL_SOCKET,  SO_REUSEADDR,
                    (char *)&on, sizeof(on));
       if (rc < 0)
    {
     perror("setsockopt() failed");
             close(listen_sd);
           exit(-1);
    }

    /************************************************/
    /* Bind the socket                              */
    /************************************************/
    memset(&addr, 0, sizeof(addr));
    addr.sin_family = AF_INET;
    addr.sin_port   = htons(SERVER_PORT);
       addr.sin_addr.s_addr = htonl(INADDR_ANY);
    rc = bind(listen_sd,
              (struct sockaddr *)&addr, sizeof(addr));
       if (rc < 0)
    {
     perror("bind() failed");
             close(listen_sd);
           exit(-1);
    }

    /************************************************/
    /* Set the listen back log                      */
    /************************************************/
    rc = listen(listen_sd, 5);
       if (rc < 0)
    {
     perror("listen() failed");
             close(listen_sd);
           exit(-1);
    }
```

```
    /*********************************************/
    /* Inform the user that the server is ready      */
    /*********************************************/
    printf("The server is ready\n");

    /*********************************************/
    /* Go through the loop once for each connection  */
    /*********************************************/
    for (i=0; i < num; i++)
    {
       /*********************************************/
    /* Wait for an incoming connection               */
       /*********************************************/
       printf("Interation: %d\n", i+1);
       printf("  waiting on accept()\n");
    accept_sd = accept(listen_sd, NULL, NULL);
    if (accept_sd < 0)
       {
       perror("accept() failed");
             close(listen_sd);
           exit(-1);
       }
       printf("  accept completed successfully\n");

       /*********************************************/
       /* Initialize the spawn parameters            */
       /*                                            */

       /* The socket descriptor for the new          */
       /* connection is mapped over to descriptor 0  */
       /* in the child program.                      */
       /*********************************************/
    memset(&inherit, 0, sizeof(inherit));
    spawn_argv[0]  = NULL;
    spawn_envp[0] = NULL;
       spawn_fdmap[0] = accept_sd;

       /*********************************************/
       /* Create the worker job                      */
       /*********************************************/
       printf("  creating worker job\n");
       pid = spawn("/QSYS.LIB/QGPL.LIB/WRKR1.PGM",
                   1, spawn_fdmap, &inherit,
                   spawn_argv, spawn_envp);
       if (pid < 0)
       {
          perror("spawn() failed");
             close(listen_sd);
       close(accept_sd);
           exit(-1);
       }
       printf("  spawn completed successfully\n");

       /*********************************************/
       /* Close down the incoming connection since   */
       /* it has been given to a worker to handle    */
       /*********************************************/
       close(accept_sd);
```

```
    }

    /*********************************************/
    /* Close down the listen socket              */
    /*********************************************/
             close(listen_sd);
}
```

소켓 설명자를 사용하여 프로세스를 완료하는 샘플 프로그램은 예: 작업자 작업이 자료 버퍼를 수신할 수 있도록 허용을 참조하십시오.

예: 작업자 작업이 자료 버퍼를 수신할 수 있도록 하기

이 예에는 작업자 작업이 클라이언트 작업에서 자료 버퍼를 수신하여 이를 다시 반향시킬 수 있게 하는 코드가 있습니다. 코드 예를 사용하는 것에 대한 정보는 코드 면책사항을 참조하십시오.

```
/*****************************************************************************/
/* Worker job that receives and echoes back a data buffer to a client     */
/*****************************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <sys/socket.h>

main (int argc, char *argv[])
{
   int    rc, len;
   int    sockfd;
   char   buffer[80];

   /*********************************************/
   /* The descriptor for the incoming connection is */
   /* passed to this worker job as a descriptor 0.  */
   /*********************************************/
   sockfd = 0;

   /*********************************************/
   /* Receive a message from the client         */
   /*********************************************/
   printf("Wait for client to send us a message\n");
      rc = recv(sockfd, buffer, sizeof(buffer), 0);
   if (rc <= 0)
   {
      perror("recv() failed");
         close(sockfd);
          exit(-1);
   }
   printf("<%s>\n", buffer);

   /*********************************************/
   /* Echo the data back to the client          */
   /*********************************************/
   printf("Echo it back\n");
   len = rc;
   rc = send(sockfd, buffer, len, 0);
   if (rc <= 0)
   {
      perror("send() failed");
```

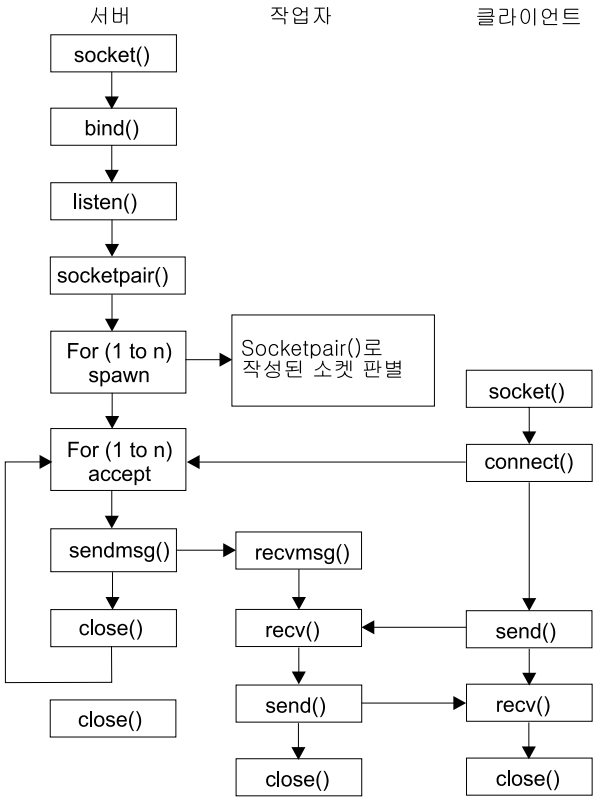
```
          close(sockfd);
           exit(-1);
    }

    /*************************************************/
    /* Close down the incoming connection            */
    /*************************************************/
          close(sockfd);

}
```

예: 프로세스간의 설명자 전달

sendmsg() 및 **recvmsg()** 예는 수신 연결을 처리하기 위해 이러한 API를 사용하는 서버 프로그램 설계 방법을 보여줍니다. 서버가 시작할 때 작업자 작업 풀을 작성합니다. 이러한 사전 할당된(spawn) 작업자 작업은 필요할 때까지 대기합니다. 클라이언트 작업이 서버로 연결될 때 서버는 작업자 작업 중 하나에 수신 연결을 제공합니다.

다음 그림은 시스템이 **sendmsg()** 및 **recvmsg()** 서버를 사용할 때 서버, 작업자 및 클라이언트 작업이 상호 작용하는 방법을 보여줍니다.

주: 이 다이어그램 중 클라이언트 부분에 대한 설명은 예: 총칭 클라이언트를 참조하십시오.



소켓 이벤트의 흐름: sendmsg() 및 recvmsg() 함수를 사용하는 서버

다음 소켓 호출 순서는 그래픽에 대한 설명을 제공합니다. 서버와 작업자 예 사이의 관계도 설명합니다. 각각의 흐름 세트에는 특정 API에서의 사용법 노트로의 링크가 포함되어 있습니다. 특정 API 사용에 대한 추가

정보가 필요하면 이런 링크를 사용할 수 있습니다. 예: sendmsg() 및 recvmsg()에 사용된 서버 프로그램에서는 **sendmsg()** 및 **recvmsg()** 기능 호출로 하위 프로세스를 작성하기 위해 다음 소켓 호출을 사용합니다.

1. **socket()** 함수는 종료점을 표시하는 소켓 설명자를 리턴합니다. 이 명령문은 TCP 전송(SOCK_STREAM)을 사용하는 INET(인터넷 프로토콜) 주소 패밀리가 이 소켓에 사용된다는 것도 식별합니다.
2. 소켓 설명자가 작성된 후에 **bind()** 함수는 소켓에 고유한 이름을 제공합니다.
3. **listen()**은 서버가 수신 클라이언트 연결을 허용할 수 있도록 합니다.
4. **socketpair()** 함수는 UNIX 데이터그램 소켓 쌍을 작성합니다. 서버는 **socketpair()** API를 사용하여 AF_UNIX 소켓 쌍을 작성할 수 있습니다.
5. **spawn()** 함수는 작업자 작업에 대한 매개변수를 초기화하여 수신 요구를 처리합니다. 이 예에서는 작성된 하위 작업이 **socketpair()**로 작성된 소켓 설명자를 계승합니다.
6. 서버에서는 수신 연결 요구를 승인하기 위해 **accept()** 함수를 사용합니다. **accept()** 호출은 수신 연결이 도착하길 기다리며 무기한 블록화됩니다.
7. **sendmsg()** 함수는 작업자 작업 중 하나로 수신 연결을 송신합니다. 하위 프로세스는 **recvmsg()** 함수를 사용하여 연결을 허용합니다. 하위 작업은 서버가 **sendmsg()**를 호출할 때 활동 상태가 아니었습니다.
8. 이 예에서 첫 번째 **close()** 함수는 허용된 소켓을 닫습니다. 두 번째 **close()** 호출은 청취 소켓을 종료합니다.

이벤트의 소켓 흐름: recvmsg()를 사용하는 작업자 작업

예: sendmsg () 및 recvmsg ()에 사용된 작업자 프로그램에서는 다음 기능 호출 순서를 사용합니다.

1. 서버가 연결을 허용하고 작업자 작업에 소켓 설명자를 전달한 후에 **recvmsg()** 함수는 설명자를 수신합니다. 이 예에서 **recvmsg()** 함수는 서버가 설명자를 송신할 때까지 기다립니다.
2. **recv()** 함수는 클라이언트로부터 메세지를 수신합니다.
3. **send()** 함수는 클라이언트로 자료를 다시 에코우합니다.
4. **close()** 함수는 작업자 작업을 종료합니다.

예: sendmsg() 및 recvmsg()에 사용된 서버 프로그램

다음 예에서는 **sendmsg()** API를 사용하여 작업자 작업 풀을 작성하는 방법을 보여줍니다. 이 예에 사용할 수 있는 공통 클라이언트 작업의 코드가 들어 있는 예를 보려면, 예: 총칭 클라이언트를 참조하십시오. 코드 예를 사용하는 것에 대한 정보는 코드 면책사항을 참조하십시오.

```
/*********************************************************************/
/* Server example that uses sendmsg() to create worker jobs          */
/*********************************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <spawn.h>

#define SERVER_PORT 12345

main (int argc, char *argv[])
{
```

```
|     int    i, num, pid, rc, on = 1;
|     int    listen_sd, accept_sd;
|     int    server_sd, worker_sd, pair_sd[2];
|     int    spawn_fdmap[1];
|     char  *spawn_argv[1];
|     char  *spawn_envp[1];
|     struct inheritance   inherit;
|  struct msghdr  msg;
|     struct sockaddr_in   addr;
|
|     /*************************************************/
|     /* If an argument was specified, use it to       */
|     /* control the number of incoming connections    */
|     /*************************************************/
|     if (argc >= 2)
|        num = atoi(argv[1]);
|     else
|        num = 1;
|
|     /*************************************************/
|     /* Create an AF_INET stream socket to receive    */
|     /* incoming connections on                       */
|     /*************************************************/
|     listen_sd = socket(AF_INET, SOCK_STREAM, 0);
|     if (listen_sd < 0)
|     {
|      perror("socket() failed");
|            exit(-1);
|     }
|
|     /*************************************************/
|    /* Allow socket descriptor to be reuseable */
|     /*************************************************/
|     rc = setsockopt(listen_sd,
|                     SOL_SOCKET,  SO_REUSEADDR,
|                     (char *)&on, sizeof(on));
|        if (rc < 0)
|     {
|      perror("setsockopt() failed");
|              close(listen_sd);
|            exit(-1);
|     }
|
|     /*************************************************/
|     /* Bind the socket                               */
|     /*************************************************/
|     memset(&addr, 0, sizeof(addr));
|     addr.sin_family = AF_INET;
|        addr.sin_addr.s_addr = htonl(INADDR_ANY);
|     addr.sin_port   = htons(SERVER_PORT);
|     rc = bind(listen_sd,
|               (struct sockaddr *)&addr, sizeof(addr));
|        if (rc < 0)
|     {
|      perror("bind() failed");
|              close(listen_sd);
|            exit(-1);
|     }
```

```
    /*************************************************/
    /* Set the listen back log                       */
    /*************************************************/
    rc = listen(listen_sd, 5);
       if (rc < 0)
    {
     perror("listen() failed");
             close(listen_sd);
           exit(-1);
    }

    /*************************************************/
    /* Create a pair of UNIX datagram sockets        */
    /*************************************************/
    rc = socketpair(AF_UNIX, SOCK_DGRAM, 0, pair_sd);
   if (rc != 0)
    {
       perror("socketpair() failed");
             close(listen_sd);
           exit(-1);
    }
    server_sd = pair_sd[0];
    worker_sd = pair_sd[1];

    /*************************************************/
    /* Initialize parms prior to entering for loop   */
    /*                                               */
    /* The worker socket descriptor is mapped to     */
    /* descriptor 0 in the child program.            */
    /*************************************************/
    memset(&inherit, 0, sizeof(inherit));
    spawn_argv[0]  = NULL;
    spawn_envp[0] = NULL;
    spawn_fdmap[0] = worker_sd;

    /*************************************************/
    /* Create each of the worker jobs                */
    /*************************************************/
    printf("Creating worker jobs...\n");
    for (i=0; i < num; i++)
    {
       pid = spawn("/QSYS.LIB/QGPL.LIB/WRKR2.PGM",
                   1, spawn_fdmap, &inherit,
                   spawn_argv, spawn_envp);
       if (pid < 0)
       {
          perror("spawn() failed");
             close(listen_sd);
    close(server_sd);
       close(worker_sd);
           exit(-1);
       }
       printf("  Worker = %d\n", pid);
    }

    /*************************************************/
    /* Close down the worker side of the socketpair  */
```

```
    /*********************************************/
       close(worker_sd);

    /*********************************************/
    /* Inform the user that the server is ready  */
    /*********************************************/
    printf("The server is ready\n");

    /*********************************************/
    /* Go through the loop once for each connection  */
    /*********************************************/
    for (i=0; i < num; i++)
    {
       /*********************************************/
    /* Wait for an incoming connection           */
       /*********************************************/
       printf("Interation: %d\n", i+1);
       printf("  waiting on accept()\n");
    accept_sd = accept(listen_sd, NULL, NULL);
    if (accept_sd < 0)
       {
       perror("accept() failed");
             close(listen_sd);
    close(server_sd);
           exit(-1);
       }
       printf("  accept completed successfully\n");

       /*********************************************/
    /* Initialize message header structure       */
       /*********************************************/
    memset(&msg,   0, sizeof(msg));

       /*********************************************/
       /* We are not sending any data so we do not  */
       /* need to set either of the msg_iov fields. */
       /* The memset of the message header structure */
       /* will set the msg_iov pointer to NULL and  */
       /* it will set the msg_iovcnt field to 0.    */
       /*********************************************/

       /*********************************************/
       /* The only fields in the message header     */
       /* structure that need to be filled in are   */
       /* the msg_accrights fields.                 */
       /*********************************************/
       msg.msg_accrights  = (char *)&accept_sd;
       msg.msg_accrightslen = sizeof(accept_sd);

       /*********************************************/
       /* Give the incoming connection to one of the */
       /* worker jobs.                              */
       /*                                           */
       /* NOTE: We do not know which worker job will */
       /*       get this inbound connection.        */
       /*********************************************/
       rc = sendmsg(server_sd, &msg, 0);
       if (rc < 0)
```

```
      {
         perror("sendmsg() failed");
            close(listen_sd);
      close(accept_sd);
   close(server_sd);
          exit(-1);
      }
      printf("  sendmsg completed successfully\n");

      /*********************************************/
      /* Close down the incoming connection since  */
      /* it has been given to a worker to handle   */
      /*********************************************/
      close(accept_sd);
   }

   /*********************************************/
   /* Close down the server and listen sockets  */
   /*********************************************/
   close(server_sd);
            close(listen_sd);
}
```

예: sendmsg () 및 recvmsg ()에 사용된 작업자 프로그램

다음 예에서는 **recvmsg()** API를 사용하여 작업자 작업을 수신하는 방법을 보여줍니다. 코드 예를 사용하는 것에 대한 정보는 코드 면책사항을 참조하십시오.

```
/*****************************************************************************/
/* Worker job that uses the recvmsg to process client requests               */
/*****************************************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <sys/socket.h>

main (int argc, char *argv[])
{
   int    rc, len;
   int    worker_sd, pass_sd;
   char   buffer[80];
struct iovec   iov[1];
struct msghdr  msg;

   /*********************************************/
   /* One of the socket descriptors that was    */
   /* returned by socketpair(), is passed to this   */
   /* worker job as descriptor 0.               */
   /*********************************************/
   worker_sd = 0;

   /*********************************************/
   /* Initialize message header structure       */
   /*********************************************/
   memset(&msg,   0, sizeof(msg));
   memset(iov,    0, sizeof(iov));

   /*********************************************/
   /* The recvmsg() call will NOT block unless a    */
```

```
    /* non-zero length data buffer is specified      */
    /************************************************/
 iov[0].iov_base = buffer;
    iov[0].iov_len  = sizeof(buffer);
 msg.msg_iov     = iov;
 msg.msg_iovlen  = 1;

    /************************************************/
    /* Fill in the msg_accrights fields so that we  */
    /* can receive the descriptor                   */
    /************************************************/
    msg.msg_accrights    = (char *)&pass_sd;
    msg.msg_accrightslen = sizeof(pass_sd);

    /************************************************/
    /* Wait for the descriptor to arrive            */
    /************************************************/
    printf("Waiting on recvmsg\n");
    rc = recvmsg(worker_sd, &msg, 0);
       if (rc < 0)
    {
       perror("recvmsg() failed");
       close(worker_sd);
           exit(-1);
    }
    else if (msg.msg_accrightslen <= 0)
    {
       printf("Descriptor was not received\n");
       close(worker_sd);
           exit(-1);
    }
    else
    {
       printf("Received descriptor = %d\n", pass_sd);
    }

    /************************************************/
    /* Receive a message from the client            */
    /************************************************/
    printf("Wait for client to send us a message\n");
    rc = recv(pass_sd, buffer, sizeof(buffer), 0);
    if (rc <= 0)
    {
       perror("recv() failed");
       close(worker_sd);
    close(pass_sd);
           exit(-1);
    }
    printf("<%s>\n", buffer);

    /************************************************/
    /* Echo the data back to the client             */
    /************************************************/
    printf("Echo it back\n");
    len = rc;
    rc = send(pass_sd, buffer, len, 0);
    if (rc <= 0)
    {
```

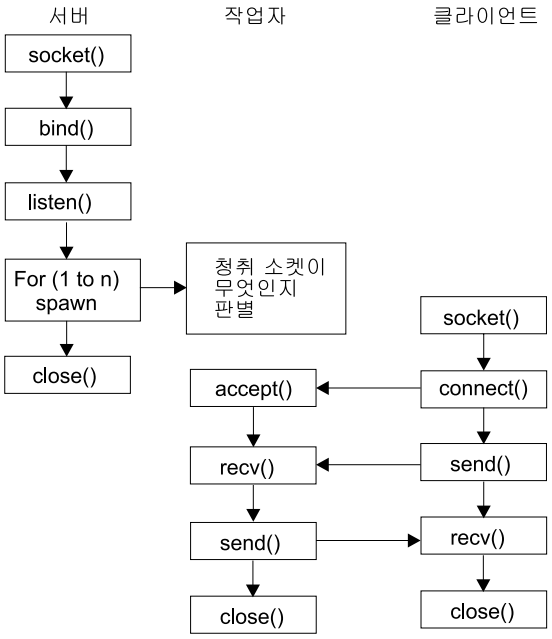
```
      perror("send() failed");
      close(worker_sd);
   close(pass_sd);
          exit(-1);
   }

   /*********************************************/
   /* Close down the descriptors                */
   /*********************************************/
      close(worker_sd);
   close(pass_sd);
}
```

예: 수신 요구 처리에 복수 accept() API 사용

다음 예에서는 수신 연결 요구를 처리하기 위해 복수 **accept()** 모델을 사용하는 서버 프로그램 설계 방식을 보여줍니다. 복수 **accept()** 서버가 시작될 때 **socket()**, **bind()** 및 **listen()**을 정상적으로 수행합니다. 그런 다음, 작업자 작업 풀을 작성하고 각 작업자 작업에 청취(listening) 소켓을 제공합니다. 그런 후 각 복수 **accept()** 작업자는 **accept()**를 호출합니다.

다음 그림은 시스템이 복수 **accept()** 서버 설계를 사용할 때 서버, 작업자 및 클라이언트 작업이 상호작용하는 방법을 보여줍니다.

주: 이 다이어그램 중 클라이언트 부분에 대한 설명은 예: 총칭 클라이언트를 참조하십시오.



소켓 이벤트의 흐름: 복수 accept() 작업자 작업의 풀을 작성하는 서버

다음 소켓 호출 순서는 그래픽에 대한 설명을 제공합니다. 서버와 작업자 예 사이의 관계도 설명합니다. 각각의 흐름 세트에는 특정 API에서의 사용법 노트로의 링크가 포함되어 있습니다. 특정 API 사용에 대한 추가 정보가 필요하면 이런 링크를 사용할 수 있습니다. 예: 복수 accept() 작업자 작업의 풀을 작성하는 서버 프로그램에서는 다음 소켓 호출을 사용하여 하위 프로세스를 작성합니다.

1. **socket()** 함수는 종료점을 표시하는 소켓 설명자를 리턴합니다. 이 명령문은 TCP 전송(SOCK_STREAM)을 사용하는 INET(인터넷 프로토콜) 주소 패밀리가 이 소켓에 사용된다는 것도 식별합니다.
2. 소켓 설명자가 작성된 후에 **bind()** 함수는 소켓에 고유한 이름을 제공합니다.
3. **listen()**은 서버가 수신 클라이언트 연결을 허용할 수 있도록 합니다.
4. **spawn()** 함수는 각각의 작업자 작업을 작성합니다.
5. 이 예에서 첫 번째 **close()** 함수는 청취 소켓을 닫습니다.

이벤트의 소켓 흐름: 복수 accept()의 작업자 작업

예: 복수 accept()의 작업자 작업에서는 다음 기능 호출 순서를 사용합니다.

1. 서버가 작업자 작업을 파생(spawn)한 후에 청취 설명자 설명자가 명령행 매개변수로 이 작업자 작업에 전달됩니다. **accept()** 함수는 수신 클라이언트 연결을 기다립니다.
2. **recv()** 함수는 클라이언트로부터 메세지를 수신합니다.
3. **send()** 함수는 클라이언트로 자료를 다시 에코우합니다.
4. **close()** 함수는 작업자 작업을 종료합니다.

예: 복수 accept() 작업자 작업의 풀을 작성할 서버 프로그램

다음 예에서는 다중 **accept()** API 모델을 사용하여 작업자 작업 풀을 작성하는 방법을 보여줍니다. 이 예에 사용할 수 있는 공통 클라이언트 작업의 코드가 들어 있는 예를 보려면, 예: 총칭 클라이언트를 참조하십시오. 코드 예를 사용하는 것에 대한 정보는 코드 면책사항을 참조하십시오.

```
/*****************************************************************************/
/* Server example creates a pool of worker jobs with multiple accept()    */
/*****************************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <spawn.h>

#define SERVER_PORT 12345

main (int argc, char *argv[])
{
   int    i, num, pid, rc, on = 1;
   int    listen_sd, accept_sd;
   int    spawn_fdmap[1];
   char  *spawn_argv[1];
   char  *spawn_envp[1];
   struct inheritance   inherit;
   struct sockaddr_in   addr;

   /*************************************************/
   /* If an argument was specified, use it to       */
   /* control the number of incoming connections    */
   /*************************************************/
   if (argc >= 2)
      num = atoi(argv[1]);
   else
```

```
|       num = 1;
|
|    /*********************************************/
|    /* Create an AF_INET stream socket to receive    */
|    /* incoming connections on                       */
|    /*********************************************/
|    listen_sd = socket(AF_INET, SOCK_STREAM, 0);
|    if (listen_sd < 0)
|    {
|     perror("socket() failed");
|           exit(-1);
|    }
|
|    /*********************************************/
|   /* Allow socket descriptor to be reuseable */
|    /*********************************************/
|    rc = setsockopt(listen_sd,
|                    SOL_SOCKET,  SO_REUSEADDR,
|                    (char *)&on, sizeof(on));
|       if (rc < 0)
|    {
|     perror("setsockopt() failed");
|             close(listen_sd);
|           exit(-1);
|    }
|
|    /*********************************************/
|    /* Bind the socket                           */
|    /*********************************************/
|    memset(&addr, 0, sizeof(addr));
|    addr.sin_family = AF_INET;
|       addr.sin_addr.s_addr = htonl(INADDR_ANY);
|    addr.sin_port   = htons(SERVER_PORT);
|    rc = bind(listen_sd,
|              (struct sockaddr *)&addr, sizeof(addr));
|       if (rc < 0)
|    {
|     perror("bind() failed");
|             close(listen_sd);
|           exit(-1);
|    }
|
|    /*********************************************/
|    /* Set the listen back log                   */
|    /*********************************************/
|    rc = listen(listen_sd, 5);
|       if (rc < 0)
|    {
|     perror("listen() failed");
|             close(listen_sd);
|           exit(-1);
|    }
|
|    /*********************************************/
|    /* Initialize parms prior to entering for loop   */
|    /*                                           */
|    /* The listen socket descriptor is mapped to     */
|    /* descriptor 0 in the child program.            */
```

```
    /*********************************************/
    memset(&inherit, 0, sizeof(inherit));
    spawn_argv[0]  = NULL;
    spawn_envp[0] = NULL;
    spawn_fdmap[0] = listen_sd;

    /*********************************************/
    /* Create each of the worker jobs            */
    /*********************************************/
    printf("Creating worker jobs...\n");
    for (i=0; i < num; i++)
    {
       pid = spawn("/QSYS.LIB/QGPL.LIB/WRKR4.PGM",
                   1, spawn_fdmap, &inherit,
                   spawn_argv, spawn_envp);
       if (pid < 0)
       {
          perror("spawn() failed");
             close(listen_sd);
           exit(-1);
       }
       printf("  Worker = %d\n", pid);
    }

    /*********************************************/
    /* Inform the user that the server is ready  */
    /*********************************************/
    printf("The server is ready\n");

    /*********************************************/
    /* Close down the listening socket           */
    /*********************************************/
             close(listen_sd);
}
```

예: 복수 accept()의 작업자 작업

이 예는 복수 **accept()** API가 작업자 작업을 수신하고 **accept()** 서버를 호출하는 방법을 보여줍니다. 코드 예를 사용하는 것에 대한 정보는 코드 면책사항을 참조하십시오.

```
/*****************************************************************************/
/* Worker job uses multiple accept() to handle incoming client connections*/
/*****************************************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <sys/socket.h>

main (int argc, char *argv[])
{
   int    rc, len;
   int    listen_sd, accept_sd;
   char   buffer[80];

   /*********************************************/
   /* The listen socket descriptor is passed to */
   /* this worker job as a command line parameter */
   /*********************************************/
   listen_sd = 0;
```

```
    /*************************************************/
    /* Wait for an incoming connection               */
    /*************************************************/
    printf("Waiting on accept()\n");
    accept_sd = accept(listen_sd, NULL, NULL);
    if (accept_sd < 0)
    {
       perror("accept() failed");
             close(listen_sd);
           exit(-1);
    }
    printf("Accept completed successfully\n");

    /*************************************************/
    /* Receive a message from the client             */
    /*************************************************/
    printf("Wait for client to send us a message\n");
    rc = recv(accept_sd, buffer, sizeof(buffer), 0);
    if (rc <= 0)
    {
       perror("recv() failed");
             close(listen_sd);
       close(accept_sd);
           exit(-1);
    }
    printf("<%s>\n", buffer);

    /*************************************************/
    /* Echo the data back to the client              */
    /*************************************************/
    printf("Echo it back\n");
    len = rc;
    rc = send(accept_sd, buffer, len, 0);
    if (rc <= 0)
    {
       perror("send() failed");
             close(listen_sd);
       close(accept_sd);
           exit(-1);
    }

    /*************************************************/
    /* Close down the descriptors                    */
    /*************************************************/
             close(listen_sd);
       close(accept_sd);
}
```

예: 총칭 클라이언트

다음 코드 예에는 공통 클라이언트 작업의 코드가 포함됩니다. 클라이언트 작업은 **socket()**, **connect()**, **send()**, **recv()** 및 **close()**를 수행합니다. 클라이언트 작업은 송신 및 수신한 자료 버퍼가 서버가 아닌 작업자 작업(worker job)으로 이동할 것이라는 것을 인식하지 못합니다. 서버가 AF_INET 주소 패밀리이건 AF_INET6 주소 패밀리이건 관계없이 작동하는 클라이언트 어플리케이션을 작성하려면, 예: IPv4 또는 IPv6 클라이언트를 사용하십시오.

| 이 클라이언트 작업은 이러한 공통 연결 지향 서버 설계 각각에 대해 작업하게 됩니다.

- 반복 서버. 샘플 프로그램은 예: 반복 서버 프로그램 작성을 참조하십시오.
- 파생(spawn) 서버 및 작업자. 샘플 프로그램은 예: spawn() API를 사용하여 하위 프로세스 작성을 참조하십시오.
- sendmsg() 서버 및 rcvmsg() 작업자. 샘플 프로그램은 예: sendmsg() 및 recvmsg()에 사용된 서버 프로그램을 참조하십시오.
- 복수 accept() 설계. 샘플 프로그램은 예: 복수 accept() 작업자 작업의 풀을 작성할 서버 프로그램을 참조하십시오.
- 비블록화 I/O 및 select() 설계. 샘플 프로그램은 예: 비블록화 I/O 및 select()를 참조하십시오.
- IPv4 또는 IPv6 클라이언트로부터 연결을 허용하는 서버. 샘플 프로그램은 예: IPv6 및 IPv4 클라이언트 모두로부터 연결 허용을 참조하십시오.

이벤트의 소켓 흐름: 충칭 클라이언트

다음 샘플 프로그램에서는 다음 기능 호출 순서를 사용합니다.

1. **socket()** 함수는 종료점을 표시하는 소켓 설명자를 리턴합니다. 이 명령문은 TCP 전송(SOCK_STREAM)을 사용하는 INET(인터넷 프로토콜) 주소 패밀리가 이 소켓에 사용된다는 것도 식별합니다.
2. 소켓 설명자가 수신된 후에 서버와의 연결 설정에 **connect()** 함수가 사용됩니다.
3. **send()** 함수는 작업자 작업으로 자료 버퍼를 송신합니다.
4. **rdcv** 함수는 작업자 작업으로부터 자료 버퍼를 수신합니다.
5. **close()** 함수는 열린 소켓 설명자를 닫습니다.

코드 예를 사용하는 것에 대한 정보는 코드 면책사항을 참조하십시오.

```
/*****************************************************************************/
/* Generic client example is used with connection-oriented server designs */
/*****************************************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <sys/socket.h>
#include <netinet/in.h>

#define SERVER_PORT 12345

main (int argc, char *argv[])
{
   int    len, rc;
   int    sockfd;
   char   send_buf[80];
   char   recv_buf[80];
   struct sockaddr_in   addr;

   /***********************************************/
   /* Create an AF_INET stream socket             */
   /***********************************************/
   sockfd = socket(AF_INET, SOCK_STREAM, 0);
   if (sockfd < 0)
   {
```

```
|       perror("socket");
|           exit(-1);
|    }
|
|    /*********************************************/
|    /* Initialize the socket address structure   */
|    /*********************************************/
|    memset(&addr, 0, sizeof(addr));
|    addr.sin_family = AF_INET;
|       addr.sin_addr.s_addr = htonl(INADDR_ANY);
|    addr.sin_port   = htons(SERVER_PORT);
|
|    /*********************************************/
|    /* Connect to the server                     */
|    /*********************************************/
|    rc = connect(sockfd,
|                 (struct sockaddr *)&addr,
|                 sizeof(struct sockaddr_in));
|       if (rc < 0)
|    {
|       perror("connect");
|          close(sockfd);
|           exit(-1);
|    }
|    printf("Connect completed.\n");
|
|    /*********************************************/
|    /* Enter data buffer that is to be sent      */
|    /*********************************************/
|    printf("Enter message to be sent:\n");
|    gets(send_buf);
|
|    /*********************************************/
|    /* Send data buffer to the worker job        */
|    /*********************************************/
|    len = send(sockfd, send_buf, strlen(send_buf) + 1, 0);
|    if (len != strlen(send_buf) + 1)
|    {
|       perror("send");
|          close(sockfd);
|           exit(-1);
|    }
|    printf("%d bytes sent\n", len);
|
|    /*********************************************/
|    /* Receive data buffer from the worker job   */
|    /*********************************************/
|    len = recv(sockfd, recv_buf, sizeof(recv_buf), 0);
|    if (len != strlen(send_buf) + 1)
|    {
|       perror("recv");
|          close(sockfd);
|           exit(-1);
|    }
|    printf("%d bytes received\n", len);
|
|    /*********************************************/
```

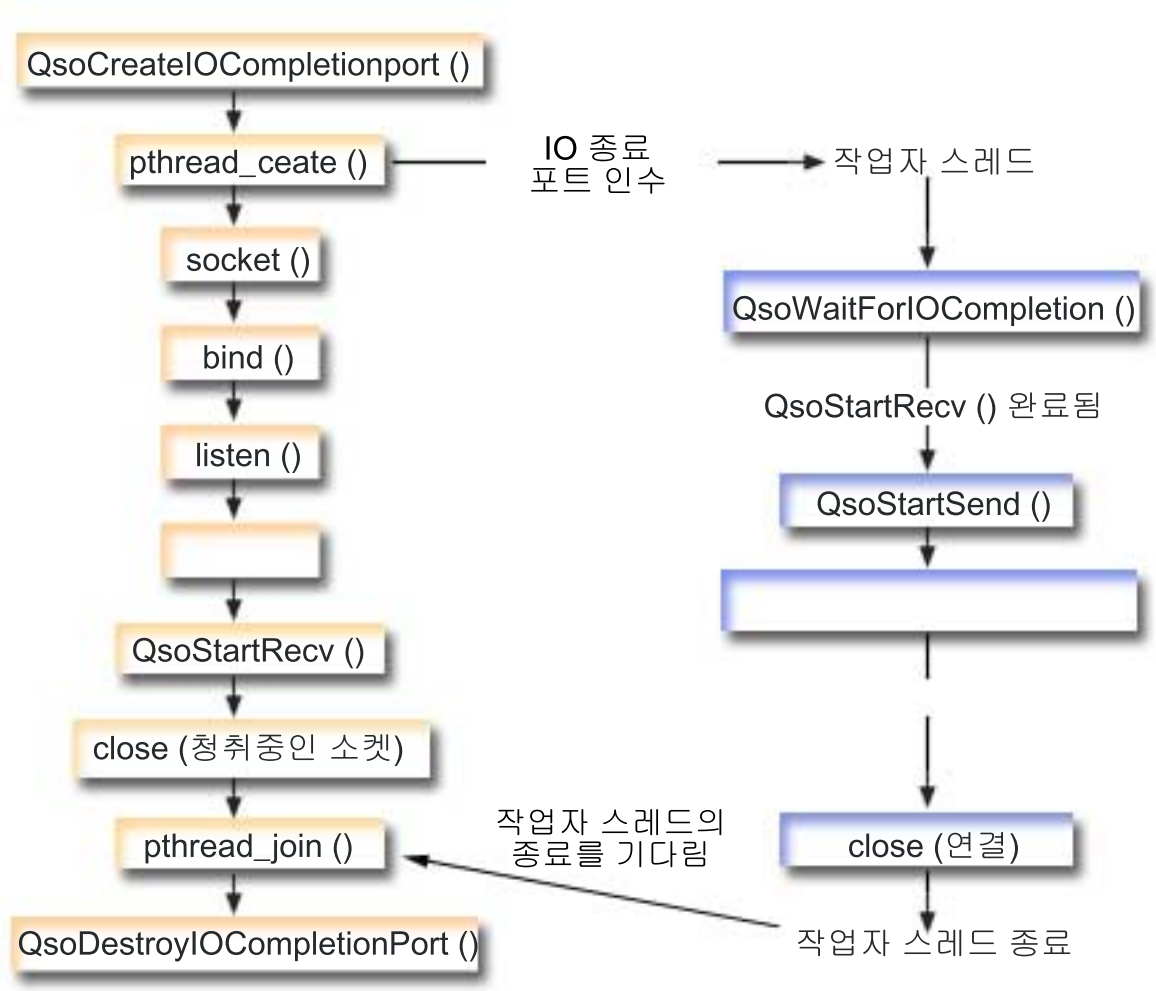
```
   /* Close down the socket                          */
   /************************************************/
         close(sockfd);
}
```

예: 비동기 I/O 사용

어플리케이션에서는 **QsoCreateIOCompletionPort()** API를 사용하여 I/O 완료 포트를 작성합니다. 이 API는 비동기 I/O 요구 완료를 스케줄하고 기다리는 데 사용될 수 있는 핸들을 리턴합니다. 어플리케이션은 I/O 완료 포트 핸들을 지정하여 입력 또는 출력 함수를 시작합니다. I/O가 완료되면 상태 정보 및 어플리케이션 정의 핸들이 지정된 I/O 완료 포트로 전송됩니다. I/O 완료 포트에 대한 전송은 대기 중인 가능한 다수의 스레드 중 하나를 정확하게 활동시킵니다. 어플리케이션에서는 다음을 수신합니다.

- 원래 요구에 제공된 버퍼
- 해당 버퍼 사이에서 처리된 자료 길이
- I/O 조작이 완료된 유형 표시
- 초기 I/O 요구에서 전달된 어플리케이션 정의 핸들

이 어플리케이션 핸들은 단순히 클라이언트 연결을 식별하는 소켓 설명자나 클라이언트 연결 상태에 대한 확장 정보를 포함하는 기억장치에 대한 포인터가 될 수 있습니다. 조작이 완료되었고 어플리케이션 핸들이 전달되었으므로, 작업자 스레드는 클라이언트 연결을 완료할 다음 단계를 판별합니다. 이러한 완료된 비동기 조작을 처리하는 작업자 스레드는 다수의 여러 클라이언트 요구를 처리할 수 있으며 하나만 결합되지 않습니다. 사용자 버퍼 사이의 복사가 서버 프로세스에 비동기적으로 발생하므로 클라이언트 요구의 대기 시간이 줄어듭니다. 이는 복수 프로세서가 있는 시스템에서 유익합니다.



소켓 이벤트의 흐름: 비동기 I/O 서버

다음 소켓 호출 순서는 그래픽에 대한 설명을 제공합니다. 서버와 작업자 예 사이의 관계도 설명합니다. 각각의 흐름 세트에는 특정 API에서의 사용법 노트로의 링크가 포함되어 있습니다. 특정 API 사용에 대한 추가 정보가 필요하면 이런 링크를 사용할 수 있습니다. 이 흐름은 다음 샘플 어플리케이션에서 소켓 호출을 설명합니다. 이 서버 예를 총칭 클라이언트 예와 함께 사용하십시오.

1. 마스터 스레드는 **QsoCreateIOCompletionPort()**를 호출하여 I/O 완료 포트를 작성합니다.
2. 마스터 스레드는 **pthread_create** 함수를 사용하여 I/O 완료 포트 요구를 처리하기 위해 작업자 스레드 풀을 작성합니다.
3. 작업자 스레드는 클라이언트 요구 처리를 기다리는 **QsoWaitForIOCompletionPort()**를 호출합니다.
4. 마스터 스레드는 클라이언트 연결을 승인한 후 작업자 스레드가 대기 중인 I/O 완료 포트를 지정하는 **QsoStartRecv()** 발행을 진행합니다.

   주: 또한 **QsoStartAccept()**를 사용하여 비동기적으로 승인을 사용할 수도 있습니다.

5. 어느 지점에서 클라이언트 요구는 서버 프로세스에 비동기적으로 도달합니다. 소켓 오퍼레이팅 시스템에서는 제공된 사용자 버퍼를 로드하고 완료된 **QsoStartRecv()** 요청을 지정된 I/O 완료 포트로 송신합니다. 한 작업자 스레드는 이 요구를 처리하기 위해 활성화되어 진행됩니다.
6. 작업자 스레드는 어플리케이션 정의 핸들에서 클라이언트 소켓 설명자를 추출하여 **QsoStartSend()** 조작을 수행하여 클라이언트로 다시 수신된 자료 반향을 진행합니다.
7. 자료가 즉시 송신되면 **QsoStartSend()**는 사실 표시를 리턴하며 그렇지 않은 경우에는 소켓 오퍼레이팅 시스템이 가능한한 곧 자료를 송신하여 지정된 I/O 완료 포트에 사실 표시를 전송합니다. 작업자 스레드가 송신된 자료 표시를 받으며 다른 요구의 I/O 완료 포트에 대기하거나 종료하도록 지시된 경우에는 종료할 수 있습니다. **QsoPostIOCompletion()**은 작업자 스레드 종료 이벤트를 전송하기 위해 마스터 스레드에서 사용할 수 있습니다.
8. 마스터 스레드는 작업자 스레드 완료를 기다린 후 **QsoDestroyIOCompletionPort()**를 호출하여 I/O 완료 포트를 손상시킵니다.

주: 공통 클라이언트 코드에 대한 이 서버 작업 예는 예: 총칭 클라이언트에 설명되어 있습니다.

이 예에서는 서버 프로그램에서 비동기 API를 사용하는 방법을 보여줍니다. 코드 예를 사용하는 것에 대한 정보는 코드 면책사항을 참조하십시오.

```
#include <stdio.h>
#include <stdlib.h>
 #include <string.h>
 #include <sys/time.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <errno.h>
 #include <unistd.h>
#define _MULTI_THREADED
 #include "pthread.h"
 #include "qsoasync.h"
 #define BufferLength 80
 #define Failure 0
 #define Success 1
 #define SERVPORT 12345

 void *workerThread(void *arg);

/*********************************************************************/
/*                                                                   */
/* Function Name: main                                               */
/*                                                                   */
/* Descriptive Name: Master thread will establish a client           */
/* connection and hand processing responsibility        */
/* to a worker thread.                                               */
 /* Note: Due to the thread attribute of this program, spawn() must  */
/* be used to invoke.                                                */
/*********************************************************************/

int main() {
 int listen_sd, client_sd, rc;
 int on = 1, ioCompPort;
```

```
pthread_t thr;
void *status;
char buffer[BufferLength];
 struct sockaddr_in serveraddr;
Qso_OverlappedIO_t ioStruct;


      /*********************************************/
/* Create an I/O completion port for this   */
   /* process.                        */
      /*********************************************/
   if ((ioCompPort = QsoCreateIOCompletionPort()) < 0)
      {
perror("QsoCreateIOCompletionPort() failed");
         exit(-1);
      }

      /*********************************************/
      /* Create a worker thread to                 */
/* to process all client requests. The   */
/* worker thread will wait for client    */
/* requests to arrive on the I/O completion  */
/* port just created.         */
      /*********************************************/
      rc = pthread_create(&thr, NULL, workerThread,
&ioCompPort);
     if (rc < 0)
      {
perror("pthread_create() failed");
QsoDestroyIOCompletionPort(ioCompPort);
           close(listen_sd);
         exit(-1);
      }

      /*********************************************/
  /* Create an AF_INET stream socket to receive    */
  /* incoming connections on                       */
      /*********************************************/
if ((listen_sd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
      {
   perror("socket() failed");
QsoDestroyIOCompletionPort(ioCompPort);
         exit(-1);
      }

      /*********************************************/
 /* Allow socket descriptor to be reuseable */
      /*********************************************/
      if ((rc = setsockopt(listen_sd, SOL_SOCKET,
                  SO_REUSEADDR,
                   (char *)&on,
                           sizeof(on))) < 0)
      {
   perror("setsockopt() failed");
QsoDestroyIOCompletionPort(ioCompPort);
           close(listen_sd);
         exit(-1);
      }
```

```
      /*******************************************/
/* bind the socket */
      /*******************************************/
 memset(&serveraddr, 0x00, sizeof(struct sockaddr_in));
 serveraddr.sin_family      = AF_INET;
 serveraddr.sin_port        = htons(SERVPORT);
 serveraddr.sin_addr.s_addr = htonl(INADDR_ANY);

      if ((rc = bind(listen_sd,
             (struct sockaddr *)&serveraddr,
                    sizeof(serveraddr))) < 0)
      {
   perror("bind() failed");
QsoDestroyIOCompletionPort(ioCompPort);
           close(listen_sd);
         exit(-1);
      }

      /*******************************************/
/* Set listen backlog */
      /*******************************************/
      if ((rc = listen(listen_sd, 10)) < 0)
      {
   perror("listen() failed");
QsoDestroyIOCompletionPort(ioCompPort);
           close(listen_sd);
         exit(-1);
      }

printf("Waiting for client connection.\n");

      /*******************************************/
/* accept an incoming client connection. */
      /*******************************************/
      if ((client_sd = accept(listen_sd, (struct sockaddr *)NULL,
NULL)) < 0)
      {
     perror("accept() failed");
QsoDestroyIOCompletionPort(ioCompPort);
           close(listen_sd);
         exit(-1);
      }

      /*******************************************/
/* Issue QsoStartRecv() to receive client  */
/* request.             */
/* Note:              */
/* postFlag == on denoting request should  */
      /*                posted to the I/O          */
      /*                completion port, even if   */
      /*                if request is immediately  */
      /*                available.  Worker thread  */
      /*                will process client        */
/* request.             */
      /*******************************************/

      /*******************************************/
```

```
 /* initialize Qso_OverlappedIO_t structure - */
 /* reserved fields must be hex 00's.    */
       /*******************************************/
 memset(&ioStruct, '\0', sizeof(ioStruct));

 ioStruct.buffer = buffer;
 ioStruct.bufferLength = sizeof(buffer);


       /*******************************************/
 /* Store the client descriptor in the    */
 /* Qso_OverlappedIO_t descriptorHandle field.*/
 /* This area is used to house information  */
 /* defining the state of the client     */
 /* connection. Field descriptorHandle is   */
 /* defined as a (void *) to allow the server */
 /* to address more extensive client     */
 /* connection state if needed.      */
       /*******************************************/
       *((int*)&ioStruct.descriptorHandle) = client_sd;
 ioStruct.postFlag = 1;
 ioStruct.fillBuffer = 0;

 rc = QsoStartRecv(client_sd, ioCompPort, &ioStruct);
 if (rc == -1)
       {
 perror("QsoStartRecv() failed");
 QsoDestroyIOCompletionPort(ioCompPort);
            close(listen_sd);
 close(client_sd);
          exit(-1);
       }
       /*******************************************/
       /* close the server's listening socket.      */
       /*******************************************/
            close(listen_sd);

       /*******************************************/
 /* Wait for worker thread to finish     */
 /* processing client connection.      */
       /*******************************************/
    rc = pthread_join(thr, &status);

 QsoDestroyIOCompletionPort(ioCompPort);
 if ( rc == 0 && (rc = __INT(status)) == Success)
       {
 printf("Success.\n");
       exit(0);
       }
   else
       {
 perror("pthread_join() reported failure");
          exit(-1);
       }
}
/* end workerThread */
```

```
/*********************************************************************/
/*                                                                   */
/* Function Name: workerThread                                       */
/*                                                                   */
/* Descriptive Name: Process client connection.                      */
/*********************************************************************/
 void *workerThread(void *arg) {
 struct timeval waitTime;
 int ioCompPort, clientfd;
 Qso_OverlappedIO_t ioStruct;
 int rc, tID;
 pthread_t thr;
 pthread_id_np_t t_id;
 t_id = pthread_getthreadid_np();
 tID = t_id.intId.lo;

      /*******************************************/
 /* I/O completion port is passed to this */
 /* routine. */
      /*******************************************/
 ioCompPort = *(int *)arg;

      /*******************************************/
 /* Wait on the supplied I/O completion port */
 /* for a client request. */
      /*******************************************/
 waitTime.tv_sec = 500;
 waitTime.tv_usec = 0;
 rc = QsoWaitForIOCompletion(ioCompPort, &ioStruct, &waitTime);
 if (rc == 1 && ioStruct.returnValue != -1)
      /*******************************************/
 /* Client request has been received. */
      /*******************************************/
        ;
   else
      {
  printf("QsoWaitForIOCompletion() or QsoStartRecv() failed.\n");
  perror("QsoWaitForIOCompletion() or QsoStartRecv() failed");
  return __VOID(Failure);
      }

      /*******************************************/
 /* Obtain the socket descriptor associated */
 /* with the client connection. */
      /*******************************************/
      clientfd = *((int *) &ioStruct.descriptorHandle);

      /*******************************************/
 /* Echo the data back to the client.    */
 /* Note: postFlag == 0. If write completes */
 /* immediate then indication will be    */
 /* returned, otherwise once the      */
 /* write is performed the I/O Completion   */
 /* port will be posted.         */
      /*******************************************/
 ioStruct.postFlag = 0;
 ioStruct.bufferLength = ioStruct.returnValue;
```

```
      rc = QsoStartSend(clientfd, ioCompPort, &ioStruct);

      if (rc == 0)
      /*******************************************/
      /* Operation complete  - data has been sent. */
      /*******************************************/
        ;
   else
      {
      /*******************************************/
/* Two possibilities          */
/* rc == -1             */
/* Error on function call        */
/* rc == 1             */
/* Write could not be immediately     */
/* performed. Once complete, the I/O    */
/* completion port will be posted.     */
      /*******************************************/

if (rc == -1)
       {
 printf("QsoStartSend() failed.\n");
         perror("QsoStartSend() failed");
close(clientfd);
 return __VOID(Failure);
       }
      /*******************************************/
/* Wait for operation to complete. */
      /*******************************************/
rc = QsoWaitForIOCompletion(ioCompPort, &ioStruct, &waitTime);
if (rc == 1 && ioStruct.returnValue != -1)
      /*******************************************/
/* Send sucessful. */
      /*******************************************/
         ;
   else
       {
 printf("QsoWaitForIOCompletion() or QsoStartSend() failed.\n");
 perror("QsoWaitForIOCompletion() or QsoStartSend() failed");
 return __VOID(Failure);
       }
      }
close(clientfd);
return __VOID(Success); } /* end workerThread */
```

예: 소스 연결 설정

글로벌 보안 툴킷(GSKit) API나 SSL_ API를 사용하여 보안 서버 및 클라이언트를 작성할 수 있습니다. 모든 IBM @server 플랫폼에 걸쳐 보안 연결을 제공하기 때문에 GSKit API가 선호되는 메소드입니다. SSL_API는 OS/400에만 고유합니다. 각각의 보안 소켓 API 세트는 사용자가 보안 소켓 연결을 설정할 때 오류를 식별할 수 있도록 코드를 리턴합니다. 이런 오류 메세지에 관한 정보에 액세스하는 것에 대한 세부사항은 보안 소켓 API 오류 코드 메세지를 참조하십시오.

다음 예에서는 이런 메소드를 각각 사용하여 보안 서버 및 클라이언트를 설정하는 방법을 설명합니다.

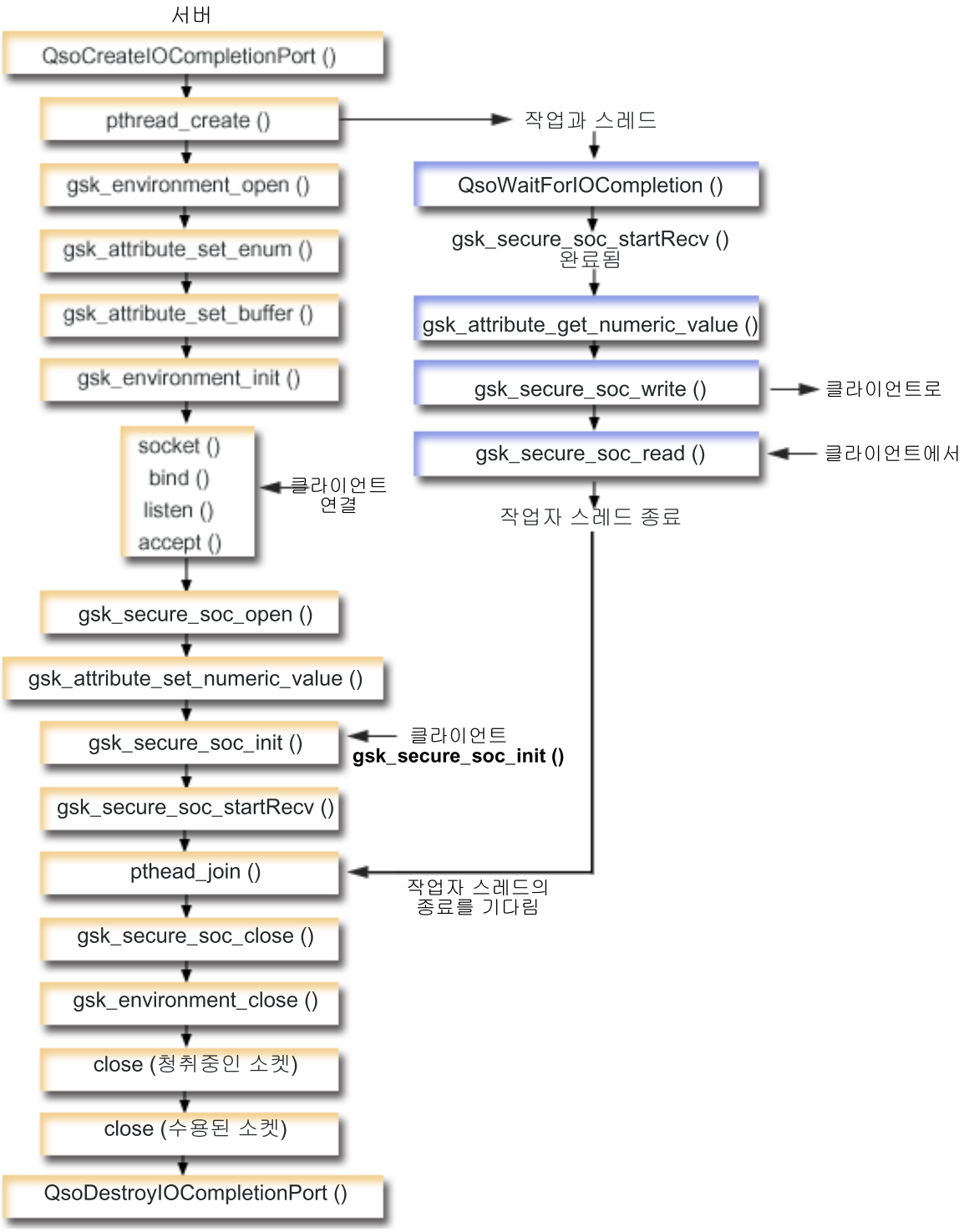
- 예: 비동기 자료 수신을 사용하는 GSKit 보안 서버
- 예: 비동기 핸드쉐이크를 사용하는 GSKit 보안 서버
- 예: GSKit API를 사용하여 보안 클라이언트 설정
- 예: SSL API를 사용하여 보안 서버 설정
- 예: SSL API를 사용하여 보안 클라이언트 설정

예: 비동기 자료 수신을 사용하는 GSKit 보안 서버

글로벌 보안 툴킷(GSKit) API를 사용하여 보안 서버를 설정하기 위해 다음 코드 예를 사용할 수 있습니다. 서버는 소켓을 열고 보안 환경을 준비하고 연결 요구를 허용하고 처리하며 클라이언트와 자료를 교환하고 세션을 종료합니다. 클라이언트도 소켓을 열고 보안 환경을 설정하고 서버를 호출하여 보안 연결을 요구하고 서버와 자료를 교환하고 세션을 종료합니다. 다음 다이어그램과 설명은 이벤트의 클라이언트/서버 흐름을 나타낸 것입니다.

주: 다음 프로그램 예에서는 AF_INET 주소 패밀리를 사용하지만 AF_INET6 주소 패밀리도 사용하도록 수정할 수 있습니다.

이벤트의 소켓 흐름: 비동기 자료 수신을 사용하는 보안 서버



이 그래픽의 클라이언트 부분을 보려면 보안 GSKit 클라이언트 그래픽을 참조하십시오.

다음 소켓 호출 순서는 그래픽에 대한 설명을 제공합니다. 서버와 클라이언트 예 사이의 관계도 설명합니다. 각각의 흐름 세트에는 특정 API에서의 사용법 노트로의 링크가 포함되어 있습니다. 특정 API 사용에 대한 추가 정보가 필요하면 이런 링크를 사용할 수 있습니다. 이 흐름은 다음 샘플 어플리케이션에서 소켓 호출을 설명합니다.

1. **QsoCreateIOCompletionPort()** 함수는 I/O 완료 포트를 작성합니다.
2. **pthread_create** 함수는 자료를 수신하고 클라이언트로 다시 자료를 에코우하기 위해 작업자 스레드를 작성합니다. 작업자 스레드는 방금 작성한 I/O 완료 포트에 클라이언트 요구가 도착하기를 기다립니다.
3. SSL 환경에 대한 핸들을 확보하기 위한 **gsk_environment_open()** 호출
4. SSL 환경의 속성을 설정하기 위한 하나 이상의 gsk_attribute_set_xxxxx() 호출. 최소한 **gsk_attribute_set_buffer()**를 호출하여 GSK_OS400_APPLICATION_ID 값이나 GSK_KEYRING_FILE 값을 설정하십시오. 이들 중 하나만 설정되어야 합니다. GSK_OS400_APPLICATION_ID 값을 사용하는 것이 더 좋습니다. 또한, **gsk_attribute_set_enum()**을 사용하여 어플리케이션 유형(클라이언트 또는 서버), GSK_SESSION_TYPE을 설정해야 합니다.
5. SSL 처리의 이 환경을 초기화하고 이 환경을 사용하여 실행할 모든 SSL 세션의 SSL 보안 정보를 설정하기 위한 **gsk_environment_init()** 호출.
6. **socket** 함수는 소켓 설명자를 작성합니다. 그러면 서버가 소켓 호출 표준 세트를 발행하는데 **bind()**, **listen()** 및 **accept()**를 발행하여 서버가 수신 연결 요구를 허용할 수 있도록 합니다.
7. **gsk_secure_soc_open()** 함수는 보안 세션의 기억장치를 확보하고 속성에 디폴트 값을 설정하며 보안 세션 관련 기능 호출에 저장되어 사용되어야 하는 핸들을 리턴합니다.
8. 보안 세션의 속성을 설정하기 위한 하나 이상의 gsk_attribute_set_xxxxx() 호출. 최소한, 이 보안 세션과 특정 소켓을 연관시키기 위한 **gsk_attribute_set_numeric_value()** 호출
9. 암호 매개변수의 SSL 핸드쉐이크 조정을 시작하기 위한 **gsk_secure_soc_init()** 호출.

   주: 일반적으로 서버 프로그램에서는 SSL 핸드쉐이크 완료에 대한 인증을 제공해야 합니다. 또한, 서버는 서버 인증과 연관된 개인 키와 인증이 저장된 키 데이터베이스 파일에 액세스할 수도 있습니다. 일부 경우 클라이언트에서는 SSL 핸드쉐이크 처리시 인증도 제공해야 합니다. 이는 클라이언트가 연결 중인 서버에 클라이언트 인증이 작동 가능한 경우에 발생합니다. **gsk_attribute_set_buffer** (GSK_OS400_APPLICATION_ID) 또는 **gsk_attribute_set_buffer** (GSK_KEYRING_FILE) API 호출은 핸드쉐이크시 사용되는 인증 및 개인 키를 확보하는 키 데이터베이스 파일을 식별합니다(유사하지 않은 방법인 경우에도).
10. **gsk_secure_soc_startRecv()** 함수는 보안 세션에서 비동기 수신 조작을 시작합니다.
11. **pthread_join**은 서버 및 작업자 프로그램을 동기화합니다. 이 함수는 스레드가 종료되기를 기다리며 스레드를 분리한 후 스레드 나감 상태를 서버에 리턴합니다.
12. **gsk_secure_soc_close()** 함수는 보안 세션을 종료합니다.
13. **gsk_environment_close()** 함수는 SSL 환경을 종료합니다.
14. **close()** 함수는 청취 소켓을 종료합니다.
15. **close()**는 허용된(클라이언트 연결) 소켓을 종료합니다.

16. **QsoDestroyIOCompletionPort()** 함수는 완료 포트를 훼손시킵니다.

이벤트의 소켓 흐름: GSKit API를 사용하는 작업자 스레드

1. 서버 어플리케이션은 작업자 스레드를 작성한 후에 서버가 작업자 스레드로 수신 클라이언트 요구를 송신하여 **gsk_secure_soc_startRecv()** 호출을 사용하여 클라이언트 자료를 처리하기를 기다립니다. **QsoWaitForIOCompletionPort()** 함수는 서버가 지정한 제공된 IO 완료 포트에서 대기합니다.
2. 클라이언트 요구가 수신되면 **gsk_attribute_get_numeric_value()** 함수가 소켓 설명자를 보안 세션과 연관시킵니다.
3. **gsk_secure_soc_write()** 함수는 보안 세션을 사용하여 클라이언트로 메세지를 송신합니다.

코드 예를 사용하는 것에 대한 정보는 코드 면책사항을 참조하십시오.

```
/* GSK Asyncronous Server Program using Application Id*/

/* "IBM grants you a nonexclusive copyright license   */
/* to use all programming code examples from which    */
/* you can generate similar function tailored to your */
/* own specific needs.                                */
/*                                                    */
/* All sample code is provided by IBM for illustrative*/
/* purposes only. These examples have not been        */
/* thoroughly tested under all conditions. IBM,       */
/* therefore, cannot guarantee or imply reliability,  */
/* serviceability, or function of these programs.     */
/*                                                    */
/* All programs contained herein are provided to you  */
/* "AS IS" without any warranties of any kind. The    */
/* implied warranties of non-infringement,            */
/* merchantability and fitness for a particular       */
/* purpose are expressly disclaimed.  "               */

/* Assummes that application id is already registered */
/* and a certificate has been associated with the     */
/* application id.                                    */
/* No parameters, some comments and many hardcoded    */
/* values to keep it short and simple                 */

/* use following command to create bound program:      */
/* CRTBNDC PGM(PROG/GSKSERVa)                         */
/*         SRCFILE(PROG/CSRC)                         */
/*         SRCMBR(GSKSERVa)                           */

#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <gskssl.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <errno.h>
#define _MULTI_THREADED
 #include "pthread.h"
 #include "qsoasync.h"
 #define Failure 0
```

```
 #define Success 1
#define TRUE             1
#define FALSE            0

 void *workerThread(void *arg);
/*********************************************************************/
/* Descriptive Name: Master thread will establish a client          */
/* connection and hand processing responsibility        */
/* to a worker thread.                                               */
 /* Note: Due to the thread attribute of this program, spawn() must  */
/* be used to invoke.                                                */
/*********************************************************************/
int main(void)
{
  gsk_handle my_env_handle=NULL;    /* secure environment handle */
  gsk_handle my_session_handle=NULL;    /* secure session handle */

    struct sockaddr_in address;
  int buf_len, on = 1, rc = 0;
  int sd = -1, lsd = -1, al = -1, ioCompPort = -1;
  int successFlag = FALSE;
    char buff[1024];
 pthread_t thr;
 void *status;
 Qso_OverlappedIO_t ioStruct;

  /*********************************************/
  /* Issue all of the command in a do/while    */
  /* loop so that clean up can happen at end   */
  /*********************************************/
   do
  {
    /*********************************************/
 /* Create an I/O completion port for this   */
    /* process.                         */
    /*********************************************/
    if ((ioCompPort = QsoCreateIOCompletionPort()) < 0)
    {
 perror("QsoCreateIOCompletionPort() failed");
        break;
    }
    /*********************************************/
 /* Create a worker thread         */
 /* to process all client requests. The   */
 /* worker thread will wait for client    */
 /* requests to arrive on the I/O completion  */
 /* port just created.         */
    /*********************************************/
    rc = pthread_create(&thr, NULL, workerThread, &ioCompPort);
      if (rc < 0)
    {
 perror("pthread_create() failed");
        break;
    }

    /* open a gsk environment */
    rc = errno = 0;
    rc = gsk_environment_open(&my_env_handle);
```

```
    if (rc != GSK_OK)
      {
        printf("gsk_environment_open() failed with rc = %d & errno = %d.\n",
               rc,errno);
          printf("rc of %d means %s\n", rc, gsk_strerror(rc));
           break;
      }

      /* set the Application ID to use */
      rc = errno = 0;
      rc = gsk_attribute_set_buffer(my_env_handle,
                                    GSK_OS400_APPLICATION_ID,
                                    "MY_SERVER_APP",
                                    13);
    if (rc != GSK_OK)
      {
        printf("gsk_attribute_set_buffer() failed with rc = %d & errno = %d.\n"
               ,rc,errno);
          printf("rc of %d means %s\n", rc, gsk_strerror(rc));
           break;
      }

      /* set this side as the server            */
      rc = errno = 0;
      rc = gsk_attribute_set_enum(my_env_handle,
                                  GSK_SESSION_TYPE,
                                  GSK_SERVER_SESSION);
    if (rc != GSK_OK)
      {
        printf("gsk_attribute_set_enum() failed with rc = %d & errno = %d.\n",
               rc,errno);
          printf("rc of %d means %s\n", rc, gsk_strerror(rc));
           break;
      }

      /* by default SSL_V2, SSL_V3, and TLS_V1 are enabled  */
      /* We will disable SSL_V2 for this example.           */
      rc = errno = 0;
      rc = gsk_attribute_set_enum(my_env_handle,
                                  GSK_PROTOCOL_SSLV2,
                                  GSK_PROTOCOL_SSLV2_OFF);
    if (rc != GSK_OK)
      {
        printf("gsk_attribute_set_enum() failed with rc = %d & errno = %d.\n",
               rc,errno);
          printf("rc of %d means %s\n", rc, gsk_strerror(rc));
           break;
      }

      /* set the cipher suite to use.  By default our default list     */
      /* of ciphers is enabled.  For this example we will just use one */
      rc = errno = 0;
      rc = gsk_attribute_set_buffer(my_env_handle,
                                    GSK_V3_CIPHER_SPECS,
                                    "05", /* SSL_RSA_WITH_RC4_128_SHA */
                                    2);
    if (rc != GSK_OK)
      {
```

```
|       printf("gsk_attribute_set_buffer() failed with rc = %d & errno = %d.\n"
|              ,rc,errno);
|         printf("rc of %d means %s\n", rc, gsk_strerror(rc));
|          break;
|     }
|
|     /* Initialize the secure environment */
|     rc = errno = 0;
|     rc = gsk_environment_init(my_env_handle);
|   if (rc != GSK_OK)
|     {
|       printf("gsk_environment_init() failed with rc = %d & errno = %d.\n",
|              rc,errno);
|         printf("rc of %d means %s\n", rc, gsk_strerror(rc));
|          break;
|     }
|
|     /* initialize a socket to be used for listening */
|     lsd = socket(AF_INET, SOCK_STREAM, 0);
|     if (lsd < 0)
|     {
|     perror("socket() failed");
|          break;
|     }
|
|     /* set socket so can be reused immediately */
|     rc = setsockopt(lsd, SOL_SOCKET,
|                     SO_REUSEADDR,
|                      (char *)&on,
|                    sizeof(on));
|       if (rc < 0)
|     {
|     perror("setsockopt() failed");
|          break;
|     }
|
|     /* bind to the local server address */
|     memset((char *) &address, 0, sizeof(address));
|     address.sin_family = AF_INET;
|     address.sin_port = 13333;
|     address.sin_addr.s_addr = 0;
|     rc = bind(lsd, (struct sockaddr *) &address, sizeof(address));
|       if (rc < 0)
|     {
|     perror("bind() failed");
|          break;
|     }
|
|     /* enable the socket for incoming client connections */
|     listen(lsd, 5);
|       if (rc < 0)
|     {
|     perror("listen() failed");
|          break;
|     }
|
|     /* accept an incoming client connection */
|     al = sizeof(address);
```

```
|      sd = accept(lsd, (struct sockaddr *) &address, &al);
|      if (sd < 0)
|      {
|        perror("accept() failed");
|           break;
|      }
|
|      /* open a secure session */
|      rc = errno = 0;
|      rc = gsk_secure_soc_open(my_env_handle, &my_session_handle);
|    if (rc != GSK_OK)
|      {
|        printf("gsk_secure_soc_open() failed with rc = %d & errno = %d.\n",
|               rc,errno);
|          printf("rc of %d means %s\n", rc, gsk_strerror(rc));
|           break;
|      }
|
|      /* associate our socket with the secure session */
|      rc=errno=0;
|      rc = gsk_attribute_set_numeric_value(my_session_handle,
|                                           GSK_FD,
|                                           sd);
|    if (rc != GSK_OK)
|      {
|        printf("gsk_attribute_set_numeric_value() failed with rc = %d ", rc);
|        printf("and errno = %d.\n", errno);
|          printf("rc of %d means %s\n", rc, gsk_strerror(rc));
|           break;
|      }
|
|      /* initiate the SSL handshake */
|      rc = errno = 0;
|      rc = gsk_secure_soc_init(my_session_handle);
|    if (rc != GSK_OK)
|      {
|        printf("gsk_secure_soc_init() failed with rc = %d & errno = %d.\n",
|               rc,errno);
|          printf("rc of %d means %s\n", rc, gsk_strerror(rc));
|           break;
|      }
|
|      /*******************************************/
|      /* Issue gsk_secure_soc_startRecv() to     */
|      /* receive client request.                 */
|   /* Note:              */
|   /* postFlag == on denoting request should  */
|   /* posted to the I/O completion port, even  */
|   /* if request is immediately available.  */
|   /* Worker thread will process client request.*/
|      /*******************************************/
|      /*******************************************/
|   /* initialize Qso_OverlappedIO_t structure - */
|   /* reserved fields must be hex 00's.    */
|      /*******************************************/
|   memset(&ioStruct, '\0', sizeof(ioStruct));
|      memset((char *) buff, 0, sizeof(buff));
|      ioStruct.buffer = buff;
```

```
    ioStruct.bufferLength = sizeof(buff);

    /*******************************************/
    /* Store the session handle in the         */
  /* Qso_OverlappedIO_t descriptorHandle field.*/
  /* This area is used to house information  */
  /* defining the state of the client     */
  /* connection. Field descriptorHandle is   */
  /* defined as a (void *) to allow the server */
  /* to address more extensive client     */
  /* connection state if needed.      */
    /*******************************************/
    ioStruct.descriptorHandle = my_session_handle;
  ioStruct.postFlag = 1;
  ioStruct.fillBuffer = 0;

    rc = gsk_secure_soc_startRecv(my_session_handle,
                                  ioCompPort,
                                  &ioStruct);
    if (rc != GSK_AS400_ASYNCHRONOUS_RECV)
    {
      printf("gsk_secure_soc_startRecv() rc = %d & errno = %d.\n",rc,errno);
        printf("rc of %d means %s\n", rc, gsk_strerror(rc));
         break;
    }

    /*******************************************/
    /* This is where the server could loop back  */
    /* to accept a new connection.             */
    /*******************************************/

    /*******************************************/
  /* Wait for worker thread to finish     */
  /* processing client connection.      */
    /*******************************************/
    rc = pthread_join(thr, &status);


    /* check status of the worker */
  if ( rc == 0 && (rc = __INT(status)) == Success)
    {
  printf("Success.\n");
     successFlag = TRUE;
    }
   else
    {
  perror("pthread_join() reported failure");
    }
   } while(FALSE);

   /* disable the SSL session */
   if (my_session_handle != NULL)
     gsk_secure_soc_close(&my_session_handle);

   /* disable the SSL environment */
   if (my_env_handle != NULL)
     gsk_environment_close(&my_env_handle);
```

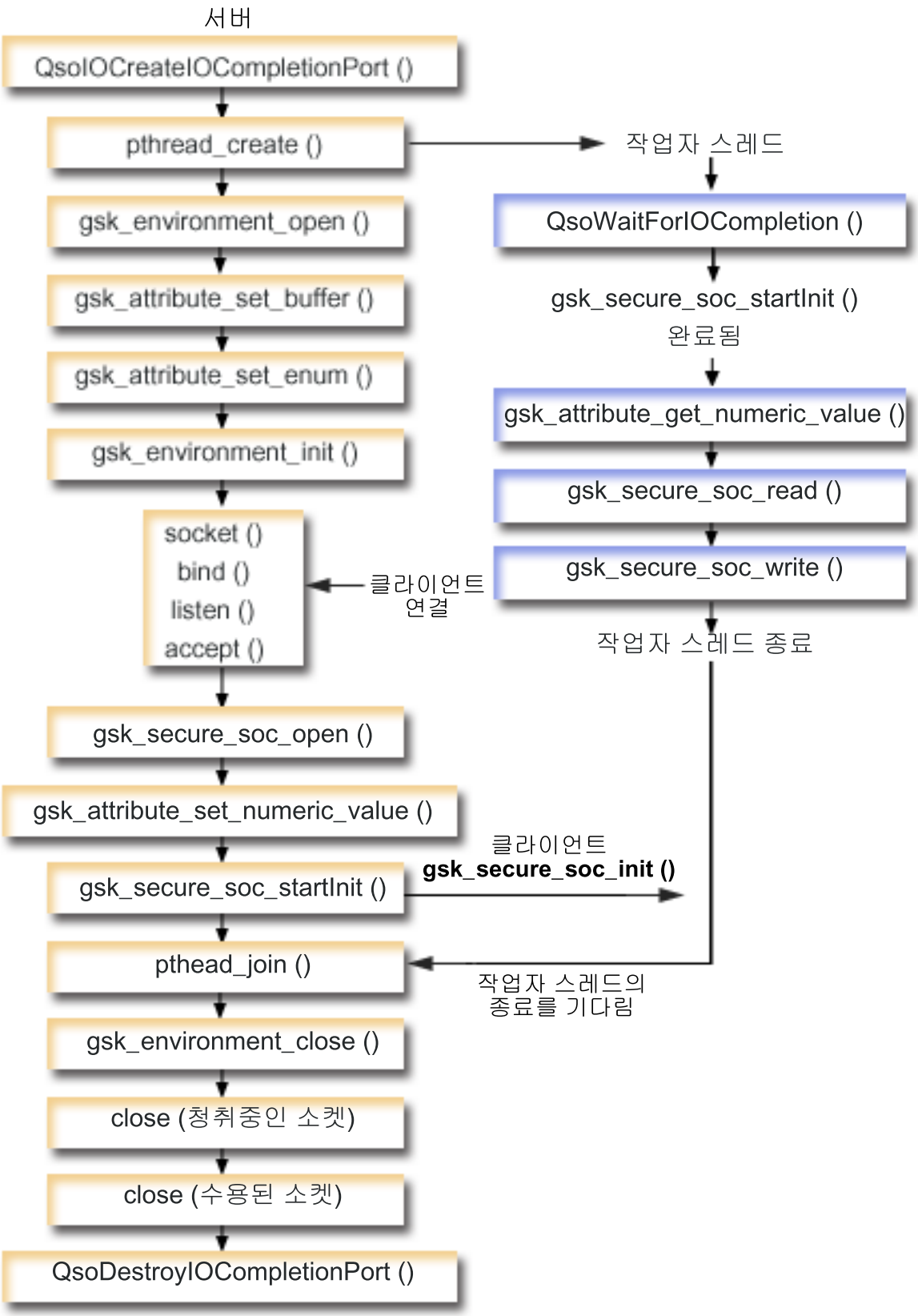
```
    /* close the listening socket */
   if (lsd > -1)
     close(lsd);
   /* close the accepted socket */
   if (sd > -1)
             close(sd);

   /* destroy the completion port */
   if (ioCompPort > -1)
  QsoDestroyIOCompletionPort(ioCompPort);

   if (successFlag)
        exit(0);
    else
          exit(-1);
 }


 /*********************************************************************/
 /* Function Name: workerThread                                       */
 /*                                                                   */
 /* Descriptive Name: Process client connection.                      */
 /*                                                                   */
 /* Note:  To make the sample more straight forward the main routine */
 /*        handles all of the clean up although this function could  */
 /*        be made responsible for the clientfd and session_handle.  */
 /*********************************************************************/
  void *workerThread(void *arg) {
  struct timeval waitTime;
     int ioCompPort = -1, clientfd = -1;
  Qso_OverlappedIO_t ioStruct;
  int rc, tID;
     int amtWritten;
     gsk_handle client_session_handle = NULL;
  pthread_t thr;
  pthread_id_np_t t_id;
  t_id = pthread_getthreadid_np();
  tID = t_id.intId.lo;
     /*********************************************/
  /* I/O completion port is passed to this */
  /* routine. */
     /*********************************************/
  ioCompPort = *(int *)arg;
     /*********************************************/
  /* Wait on the supplied I/O completion port */
  /* for a client request. */
     /*********************************************/
  waitTime.tv_sec = 500;
  waitTime.tv_usec = 0;
  rc = QsoWaitForIOCompletion(ioCompPort, &ioStruct, &waitTime);
     if ((rc == 1) &&
         (ioStruct.returnValue == GSK_OK) &&
         (ioStruct.operationCompleted == GSKSECURESOCSTARTRECV))
     /*********************************************/
  /* Client request has been received. */
     /*********************************************/
     ;
    else
```

```
|      {
|        perror("QsoWaitForIOCompletion()/gsk_secure_soc_startRecv() failed");
|        printf("ioStruct.returnValue = %d.\n", ioStruct.returnValue);
|    return __VOID(Failure);
|      }
|
|      /* write results to screen */
|      printf("gsk_secure_soc_startRecv() received %d bytes, here they are:\n",
|             ioStruct.secureDataTransferSize);
|      printf("%s\n",ioStruct.buffer);
|
|      /*******************************************/
|      /* Obtain the session handle associated      */
|    /* with the client connection. */
|      /*******************************************/
|      client_session_handle = ioStruct.descriptorHandle;
|
|      /* get the socket associatied with the secure session */
|      rc=errno=0;
|      rc = gsk_attribute_get_numeric_value(client_session_handle,
|                                           GSK_FD,
|                                           &clientfd);
|    if (rc != GSK_OK)
|      {
|        printf("gsk_attribute_get_numeric_value() rc = %d & errno = %d.\n",
|               rc,errno);
|          printf("rc of %d means %s\n", rc, gsk_strerror(rc));
|    return __VOID(Failure);
|      }
|
|      /* send the message to the client using the secure session */
|      amtWritten = 0;
|      rc = gsk_secure_soc_write(client_session_handle,
|                                ioStruct.buffer,
|                                ioStruct.secureDataTransferSize,
|                                &amtWritten);
|      if (amtWritten != ioStruct.secureDataTransferSize)
|      {
|    if (rc != GSK_OK)
|        {
|          printf("gsk_secure_soc_write() rc = %d and errno = %d.\n",
|               rc,errno);
|          printf("rc of %d means %s\n", rc, gsk_strerror(rc));
|    return __VOID(Failure);
|        }
|     else
|        {
|          printf("gsk_secure_soc_write() did not write all data.\n");
|    return __VOID(Failure);
|        }
|      }
|
|      /* write results to screen */
|      printf("gsk_secure_soc_write() wrote %d bytes...\n", amtWritten);
|      printf("%s\n",ioStruct.buffer);
|
|    return __VOID(Success); } /* end workerThread */
```

예: 비동기 핸드쉐이크를 사용하는 GSKit 보안 서버

V5R2에서는 새롭게 OS/400 소켓이 **gsk_secure_soc_startInit()** API를 채택하였습니다. 이 API를 사용하면 비동기로 요구를 처리할 수 있는 보안 서버 어플리케이션을 작성할 수 있습니다. 다음 코드 샘플은 이 API를 사용할 수 있는 방법 예를 제공합니다. 비동기 자료 수신을 사용하는 GSKit 보안 서버 예와 유사하지만 이 새로운 API를 사용하여 보안 세션을 시작합니다.

다음 그래픽은 보안 서버에서 비동기 핸드쉐이크를 협상하는 데 사용되는 기능 호출을 나타낸 것입니다.



이 그래픽의 클라이언트 부분을 보려면 GSKit 클라이언트를 참조하십시오.

이벤트의 소켓 흐름: 비동기 핸드쉐이크를 사용하는 GSKit 보안 서버

이 흐름은 다음 샘플 어플리케이션에서 소켓 호출을 설명합니다.

1. **QsoCreateIOCompletionPort()** 함수는 I/O 완료 포트를 작성합니다.
2. **pthread_create** 함수는 모든 클라이언트 요구를 처리하기 위해 작업자 스레드를 작성합니다. 작업자 스레드는 방금 작성한 I/O 완료 포트에 클라이언트 요구가 도착하기를 기다립니다.
3. SSL 환경에 대한 핸들을 확보하기 위한 **gsk_environment_open()** 호출
4. SSL 환경의 속성을 설정하기 위한 하나 이상의 gsk_attribute_set_xxxxx() 호출. 최소한 **gsk_attribute_set_buffer()**를 호출하여 GSK_OS400_APPLICATION_ID 값이나 GSK_KEYRING_FILE 값을 설정하십시오. 이들 중 하나만 설정되어야 합니다. GSK_OS400_APPLICATION_ID 값을 사용하는 것이 더 좋습니다. 또한, **gsk_attribute_set_enum()**을 사용하여 어플리케이션 유형(클라이언트 또는 서버), GSK_SESSION_TYPE을 설정해야 합니다.
5. SSL 처리의 이 환경을 초기화하고 이 환경을 사용하여 실행할 모든 SSL 세션의 SSL 보안 정보를 설정하기 위한 **gsk_environment_init()** 호출.
6. **socket** 함수는 소켓 설명자를 작성합니다. 그러면 서버가 소켓 호출 표준 세트를 발행하는데 **bind()**, **listen()** 및 **accept()**를 발행하여 서버가 수신 연결 요구를 허용할 수 있도록 합니다.
7. **gsk_secure_soc_open()** 함수는 보안 세션의 기억장치를 확보하고 속성에 디폴트 값을 설정하며 보안 세션 관련 기능 호출에 저장되어 사용되어야 하는 핸들을 리턴합니다.
8. 보안 세션의 속성을 설정하기 위한 하나 이상의 gsk_attribute_set_xxxxx() 호출. 최소한, 이 보안 세션과 특정 소켓을 연관시키기 위한 **gsk_attribute_set_numeric_value()** 호출
9. **gsk_secure_soc_startInit()** 함수는 SSL 환경 및 보안 세션에 속성 세트를 사용하여 비동기로 보안 세션 협상을 시작합니다. 여기서 제어가 프로그램에 리턴됩니다. 핸드쉐이크 처리가 완료되면 완료 포트에 결과가 게시됩니다. 스레드는 다른 처리를 계속할 수 있지만 단순화하기 위해 여기서는 작업자 스레드가 완료되기를 기다리도록 선택하였습니다.

   주: 일반적으로 서버 프로그램에서는 SSL 핸드쉐이크 완료에 대한 인증을 제공해야 합니다. 또한, 서버는 서버 인증과 연관된 개인 키와 인증이 저장된 키 데이터베이스 파일에 액세스할 수도 있습니다. 일부 경우 클라이언트에서는 SSL 핸드쉐이크 처리시 인증도 제공해야 합니다. 이는 클라이언트가 연결 중인 서버에 클라이언트 인증이 작동 가능한 경우에 발생합니다. **gsk_attribute_set_buffer** (GSK_OS400_APPLICATION_ID) 또는 **gsk_attribute_set_buffer** (GSK_KEYRING_FILE) API 호출은 핸드쉐이크시 사용되는 인증 및 개인 키를 확보하는키 데이터베이스 파일을 식별합니다(유사하지 않은 방법인 경우에도).
10. **pthread_join**은 서버 및 작업자 프로그램을 동기화합니다. 이 함수는 스레드가 종료되기를 기다리며 스레드를 분리한 후 스레드 나감 상태를 서버에 리턴합니다.
11. **gsk_secure_soc_close()** 함수는 보안 세션을 종료합니다.

12. **gsk_environment_close()** 함수는 SSL 환경을 종료합니다.
13. **close()** 함수는 청취 소켓을 종료합니다.
14. **close()**는 허용된(클라이언트 연결) 소켓을 종료합니다.
15. **QsoDestroyIOCompletionPort()** 함수는 완료 포트를 훼손시킵니다.

이벤트의 소켓 흐름: 보안 비동기 요구를 처리하는 작업자 스레드

1. 서버 어플리케이션은 작업자 스레드를 작성한 후에 서버가 작업자 스레드로 수신 클라이언트 요구를 송신하여 처리하기를 기다립니다. **QsoWaitForIOCompletionPort()** 함수는 서버가 지정한 제공된 IO 완료 포트에서 대기합니다. 이 호출은 **gsk_secure_soc_startInit()**가 완료될 때까지 기다립니다.
2. 클라이언트 요구가 수신되면 **gsk_attribute_get_numeric_value()** 함수가 소켓 설명자를 보안 세션과 연관시킵니다.
3. **gsk_secure_soc_read()** 함수는 보안 세션을 사용하여 클라이언트에서 메세지를 수신합니다.
4. **gsk_secure_soc_write()** 함수는 보안 세션을 사용하여 클라이언트로 메세지를 송신합니다.

코드 예를 사용하는 것에 대한 정보는 코드 면책사항을 참조하십시오.

```
/* GSK Asyncronous Server Program using Application Id*/
/* and gsk_secure_soc_startInit()                      */

/* Assummes that application id is already registered */
/* and a certificate has been associated with the     */
/* application id.                                    */
/* No parameters, some comments and many hardcoded    */
/* values to keep it short and simple                 */

/* use following command to create bound program:      */
/* CRTBNDC PGM(MYLIB/GSKSERVSI)                       */
/*         SRCFILE(MYLIB/CSRC)                        */
/*         SRCMBR(GSKSERVSI)                          */

#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <gskssl.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <errno.h>
#define _MULTI_THREADED
 #include "pthread.h"
 #include "qsoasync.h"
 #define Failure 0
 #define Success 1
#define TRUE             1
#define FALSE            0

 void *workerThread(void *arg);
/*****************************************************************/
/* Descriptive Name: Master thread will establish a client         */
/* connection and hand processing responsibility        */
/* to a worker thread.                                             */
```

```
 /* Note: Due to the thread attribute of this program, spawn() must  */
/* be used to invoke.                                                */
/*******************************************************************/
int main(void)
{
  gsk_handle my_env_handle=NULL;    /* secure environment handle */
  gsk_handle my_session_handle=NULL;    /* secure session handle */

    struct sockaddr_in address;
  int buf_len, on = 1, rc = 0;
  int sd = -1, lsd = -1, al, ioCompPort = -1;
  int successFlag = FALSE;
 pthread_t thr;
 void *status;
 Qso_OverlappedIO_t ioStruct;

  /*******************************************/
  /* Issue all of the command in a do/while    */
  /* loop so that clean up can happen at end   */
  /*******************************************/

   do
  {
    /*******************************************/
 /* Create an I/O completion port for this   */
    /* process.                         */
    /*******************************************/
    if ((ioCompPort = QsoCreateIOCompletionPort()) < 0)
    {
 perror("QsoCreateIOCompletionPort() failed");
        break;
    }
    /*******************************************/
 /* Create a worker thread         */
 /* to process all client requests. The   */
 /* worker thread will wait for client    */
 /* requests to arrive on the I/O completion  */
 /* port just created.         */
    /*******************************************/
    rc = pthread_create(&thr, NULL, workerThread, &ioCompPort);
      if (rc < 0)
    {
 perror("pthread_create() failed");
        break;
    }

    /* open a gsk environment */
    rc = errno = 0;
    printf("gsk_environment_open()\n");
    rc = gsk_environment_open(&my_env_handle);
  if (rc != GSK_OK)
    {
      printf("gsk_environment_open() failed with rc = %d and errno = %d.\n",
             rc,errno);
        printf("rc of %d means %s\n", rc, gsk_strerror(rc));
         break;
    }

```

```
  /* set the Application ID to use */
  rc = errno = 0;
  rc = gsk_attribute_set_buffer(my_env_handle,
                                GSK_OS400_APPLICATION_ID,
                                "MY_SERVER_APP",
                                13);
if (rc != GSK_OK)
  {
    printf("gsk_attribute_set_buffer() failed with rc = %d and errno = %d.\n"
           ,rc,errno);
      printf("rc of %d means %s\n", rc, gsk_strerror(rc));
       break;
  }

  /* set this side as the server            */
  rc = errno = 0;
  rc = gsk_attribute_set_enum(my_env_handle,
                              GSK_SESSION_TYPE,
                              GSK_SERVER_SESSION);
if (rc != GSK_OK)
  {
    printf("gsk_attribute_set_enum() failed with rc = %d and errno = %d.\n",
           rc,errno);
      printf("rc of %d means %s\n", rc, gsk_strerror(rc));
       break;
  }

  /* by default SSL_V2, SSL_V3, and TLS_V1 are enabled  */
  /* We will disable SSL_V2 for this example.           */
  rc = errno = 0;
  rc = gsk_attribute_set_enum(my_env_handle,
                              GSK_PROTOCOL_SSLV2,
                              GSK_PROTOCOL_SSLV2_OFF);
if (rc != GSK_OK)
  {
    printf("gsk_attribute_set_enum() failed with rc = %d and errno = %d.\n",
           rc,errno);
      printf("rc of %d means %s\n", rc, gsk_strerror(rc));
       break;
  }

  /* set the cipher suite to use.  By default our default list     */
  /* of ciphers is enabled.  For this example we will just use one */
  rc = errno = 0;
  rc = gsk_attribute_set_buffer(my_env_handle,
                                GSK_V3_CIPHER_SPECS,
                                "05", /* SSL_RSA_WITH_RC4_128_SHA */
                                2);
if (rc != GSK_OK)
  {
    printf("gsk_attribute_set_buffer() failed with rc = %d and errno = %d.\n"
           ,rc,errno);
      printf("rc of %d means %s\n", rc, gsk_strerror(rc));
       break;
  }

  /* Initialize the secure environment */
  rc = errno = 0;
```

```
|      printf("gsk_environment_init()\n");
|      rc = gsk_environment_init(my_env_handle);
|    if (rc != GSK_OK)
|      {
|        printf("gsk_environment_init() failed with rc = %d and errno = %d.\n",
|               rc,errno);
|          printf("rc of %d means %s\n", rc, gsk_strerror(rc));
|           break;
|      }
|
|      /* initialize a socket to be used for listening */
|      printf("socket()\n");
|      lsd = socket(AF_INET, SOCK_STREAM, 0);
|      if (lsd < 0)
|      {
|      perror("socket() failed");
|           break;
|      }
|
|      /* set socket so can be reused immediately */
|      rc = setsockopt(lsd, SOL_SOCKET,
|                      SO_REUSEADDR,
|                       (char *)&on,
|                     sizeof(on));
|        if (rc < 0)
|      {
|      perror("setsockopt() failed");
|           break;
|      }
|
|      /* bind to the local server address */
|      memset((char *) &address, 0, sizeof(address));
|      address.sin_family = AF_INET;
|      address.sin_port = 13333;
|      address.sin_addr.s_addr = 0;
|      printf("bind()\n");
|      rc = bind(lsd, (struct sockaddr *) &address, sizeof(address));
|        if (rc < 0)
|      {
|      perror("bind() failed");
|           break;
|      }
|
|      /* enable the socket for incoming client connections */
|      printf("listen()\n");
|      listen(lsd, 5);
|        if (rc < 0)
|      {
|      perror("listen() failed");
|           break;
|      }
|
|      /* accept an incoming client connection */
|      al = sizeof(address);
|      printf("accept()\n");
|      sd = accept(lsd, (struct sockaddr *) &address, &al);
|      if (sd < 0)
|      {
```

```
|       perror("accept() failed");
|          break;
|     }
|
|     /* open a secure session */
|     rc = errno = 0;
|     printf("gsk_secure_soc_open()\n");
|     rc = gsk_secure_soc_open(my_env_handle, &my_session_handle);
|   if (rc != GSK_OK)
|     {
|       printf("gsk_secure_soc_open() failed with rc = %d and errno = %d.\n",
|              rc,errno);
|         printf("rc of %d means %s\n", rc, gsk_strerror(rc));
|          break;
|     }
|     /* associate our socket with the secure session */
|     rc=errno=0;
|     rc = gsk_attribute_set_numeric_value(my_session_handle,
|                                          GSK_FD,
|                                          sd);
|   if (rc != GSK_OK)
|     {
|       printf("gsk_attribute_set_numeric_value() failed with rc = %d ", rc);
|       printf("and errno = %d.\n", errno);
|         printf("rc of %d means %s\n", rc, gsk_strerror(rc));
|          break;
|     }
|
|     /*********************************************/
|     /* Issue gsk_secure_soc_startInit() to       */
|     /* process SSL Handshake flow asyncronously  */
|     /*********************************************/
|     /*********************************************/
|  /* initialize Qso_OverlappedIO_t structure - */
|  /* reserved fields must be hex 00's.    */
|     /*********************************************/
|  memset(&ioStruct, '\0', sizeof(ioStruct));
|
|     /*********************************************/
|     /* Store the session handle in the           */
|  /* Qso_OverlappedIO_t descriptorHandle field.*/
|  /* This area is used to house information  */
|  /* defining the state of the client     */
|  /* connection. Field descriptorHandle is   */
|  /* defined as a (void *) to allow the server */
|  /* to address more extensive client     */
|  /* connection state if needed.      */
|     /*********************************************/
|     ioStruct.descriptorHandle = my_session_handle;
|
|     /* initiate the SSL handshake */
|     rc = errno = 0;
|     printf("gsk_secure_soc_startInit()\n");
|     rc = gsk_secure_soc_startInit(my_session_handle, ioCompPort, &ioStruct);
|     if (rc != GSK_OS400_ASYNCHRONOUS_SOC_INIT)
|     {
|       printf("gsk_secure_soc_startInit() rc = %d and errno = %d.\n",rc,errno);
|         printf("rc of %d means %s\n", rc, gsk_strerror(rc));
```

```
          break;
      }
     else
        printf("gsk_secure_soc_startInit got GSK_OS400_ASYNCHRONOUS_SOC_INIT\n");

      /*******************************************/
      /* This is where the server could loop back  */
      /* to accept a new connection.               */
      /*******************************************/

      /*******************************************/
   /* Wait for worker thread to finish     */
   /* processing client connection.      */
      /*******************************************/
      rc = pthread_join(thr, &status);


      /* check status of the worker */
   if ( rc == 0 && (rc = __INT(status)) == Success)
      {
   printf("Success.\n");
   printf("Success.\n");
        successFlag = TRUE;
      }
     else
      {
   perror("pthread_join() reported failure");
      }
    } while(FALSE);

    /* disable the SSL session */
    if (my_session_handle != NULL)
      gsk_secure_soc_close(&my_session_handle);

    /* disable the SSL environment */
    if (my_env_handle != NULL)
      gsk_environment_close(&my_env_handle);

      /* close the listening socket */
    if (lsd > -1)
      close(lsd);
    /* close the accepted socket */
    if (sd > -1)
              close(sd);

    /* destroy the completion port */
    if (ioCompPort > -1)
   QsoDestroyIOCompletionPort(ioCompPort);

    if (successFlag)
         exit(0);

            exit(-1);
 }


 /*********************************************************************/
 /* Function Name: workerThread                                       */
```

```
| /*                                                                  */
| /* Descriptive Name: Process client connection.                     */
| /*                                                                  */
| /* Note:  To make the sample more straight forward the main routine */
| /*        handles all of the clean up although this function could  */
| /*        be made responsible for the clientfd and session_handle.  */
| /********************************************************************/
|  void *workerThread(void *arg) {
|  struct timeval waitTime;
|  int ioCompPort, clientfd;
|  Qso_OverlappedIO_t ioStruct;
|  int rc, tID;
|   int amtWritten, amtRead;
|     char buff[1024];
|     gsk_handle client_session_handle;
|  pthread_t thr;
|  pthread_id_np_t t_id;
|  t_id = pthread_getthreadid_np();
|  tID = t_id.intId.lo;
|     /*******************************************/
|  /* I/O completion port is passed to this */
|  /* routine. */
|     /*******************************************/
|  ioCompPort = *(int *)arg;
|     /*******************************************/
|  /* Wait on the supplied I/O completion port */
|     /* for the SSL handshake to complete.      */
|     /*******************************************/
|  waitTime.tv_sec = 500;
|  waitTime.tv_usec = 0;
|
|     sleep(4);
|     printf("QsoWaitForIOCompletion()\n");
|  rc = QsoWaitForIOCompletion(ioCompPort, &ioStruct, &waitTime);
|     if ((rc == 1) &&
|         (ioStruct.returnValue == GSK_OK) &&
|         (ioStruct.operationCompleted == GSKSECURESOCSTARTINIT))
|     /*******************************************/
|     /* SSL Handshake has completed.            */
|     /*******************************************/
|     ;
|    else
|     {
|       printf("QsoWaitForIOCompletion()/gsk_secure_soc_startInit() failed.\n");
|       printf("rc == %d, returnValue - %d, operationCompleted = %d\n",
|              rc, ioStruct.returnValue, ioStruct.operationCompleted);
|       perror("QsoWaitForIOCompletion()/gsk_secure_soc_startInit() failed");
|   return __VOID(Failure);
|     }
|
|     /*******************************************/
|     /* Obtain the session handle associated    */
|  /* with the client connection. */
|     /*******************************************/
|     client_session_handle = ioStruct.descriptorHandle;
|
|     /* get the socket associatied with the secure session */
|     rc=errno=0;
```

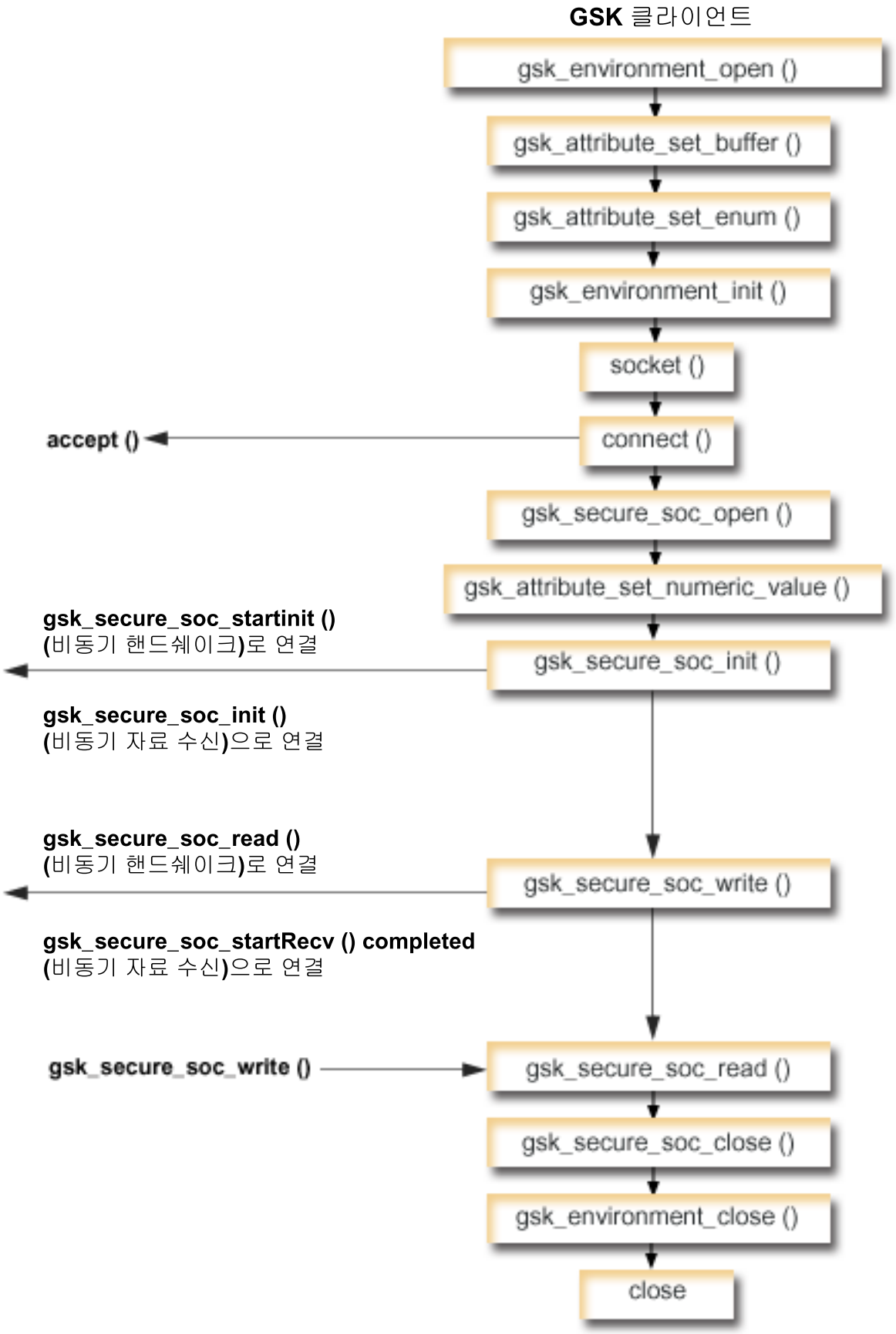
```
|      printf("gsk_attribute_get_numeric_value()\n");
|      rc = gsk_attribute_get_numeric_value(client_session_handle,
|                                           GSK_FD,
|                                           &clientfd);
|    if (rc != GSK_OK)
|      {
|        printf("gsk_attribute_get_numeric_value() rc = %d and errno = %d.\n",
|               rc,errno);
|          printf("rc of %d means %s\n", rc, gsk_strerror(rc));
|    return __VOID(Failure);
|      }
|    /* memset buffer to hex zeros                    */
|      memset((char *) buff, 0, sizeof(buff));
|      amtRead = 0;
|      /* receive a message from the client using the secure session */
|      printf("gsk_secure_soc_read()\n");
|      rc = gsk_secure_soc_read(client_session_handle,
|                               buff,
|                              sizeof(buff),
|                              &amtRead);
|
|    if (rc != GSK_OK)
|      {
|        printf("gsk_secure_soc_read() rc = %d and errno = %d.\n",rc,errno);
|          printf("rc of %d means %s\n", rc, gsk_strerror(rc));
|    return;
|      }
|
|      /* write results to screen */
|      printf("gsk_secure_soc_read() received %d bytes, here they are ...\n",
|             amtRead);
|      printf("%s\n",buff);
|
|
|      /* send the message to the client using the secure session */
|      amtWritten = 0;
|      printf("gsk_secure_soc_write()\n");
|      rc = gsk_secure_soc_write(client_session_handle,
|                               buff,
|                               amtRead,
|                               &amtWritten);
|      if (amtWritten != amtRead)
|      {
|    if (rc != GSK_OK)
|        {
|          printf("gsk_secure_soc_write() rc = %d and errno = %d.\n",rc,errno);
|          printf("rc of %d means %s\n", rc, gsk_strerror(rc));
|    return __VOID(Failure);
|        }
|     else
|        {
|          printf("gsk_secure_soc_write() did not write all data.\n");
|    return __VOID(Failure);
|        }
|      }
|      /* write results to screen */
|      printf("gsk_secure_soc_write() wrote %d bytes...\n", amtWritten);
|      printf("%s\n",buff);
```

```
    return __VOID(Success);
}
/* end workerThread */
```

예: 글로벌 보안 툴킷(GSKit) API를 사용하여 보안 클라이언트 설정

다음 코드 샘플은 GSKit API를 사용하는 클라이언트 예를 제공합니다. 코드 예를 사용하는 것에 대한 정보는 코드 면책사항을 참조하십시오.

다음 그래픽은 GSKit API를 사용하여 보안 클라이언트에서 기능 호출을 나타낸 것입니다.


GSK 클라이언트
gsk_environment_open ()
gsk_attribute_set_buffer ()
gsk_attribute_set_enum ()
gsk_environment_init ()
socket ()
connect ()
accept ()
gsk_secure_soc_open ()
gsk_attribute_set_numeric_value ()
gsk_secure_soc_startinit ()
(비동기 핸드쉐이크)로 연결
gsk_secure_soc_init ()
gsk_secure_soc_init ()
(비동기 자료 수신)으로 연결
gsk_secure_soc_read ()
(비동기 핸드쉐이크)로 연결
gsk_secure_soc_write ()
gsk_secure_soc_startRecv () completed
(비동기 자료 수신)으로 연결
gsk_secure_soc_write ()
gsk_secure_soc_read ()
gsk_secure_soc_close ()
gsk_environment_close ()
close

이벤트의 소켓 흐름: GSKit 클라이언트

이 흐름은 다음 샘플 어플리케이션에서 소켓 호출을 설명합니다. 이 클라이언트 예를 GSKit 서버 예 및 예: 비동기 핸드쉐이크를 사용하는 GSKit 보안 서버와 함께 사용하십시오.

1. **gsk_environment_open()** 함수는 SSL 환경에 대한 핸들을 확보합니다.
2. SSL 환경의 속성을 설정하기 위한 하나 이상의 gsk_attribute_set_xxxxx() 호출. 최소한 **gsk_attribute_set_buffer()**를 호출하여 GSK_OS400_APPLICATION_ID 값이나 GSK_KEYRING_FILE 값을 설정하십시오. 이들 중 하나만 설정되어야 합니다. GSK_OS400_APPLICATION_ID 값을 사용하는 것이 더 좋습니다. 또한, **gsk_attribute_set_enum()**을 사용하여 어플리케이션 유형(클라이언트 또는 서버), GSK_SESSION_TYPE을 설정해야 합니다.
3. SSL 처리의 이 환경을 초기화하고 이 환경을 사용하여 실행할 모든 SSL 세션의 SSL 보안 정보를 설정하기 위한 **gsk_environment_init()** 호출.
4. **socket** 함수는 소켓 설명자를 작성합니다. 그러면 클라이언트가 **connect()**를 발행하여 서버 어플리케이션에 연결합니다.
5. **gsk_secure_soc_open()** 함수는 보안 세션의 기억장치를 확보하고 속성에 디폴트 값을 설정하며 보안 세션 관련 기능 호출에 저장되어 사용되어야 하는 핸들을 리턴합니다.
6. **gsk_attribute_set_numeric_value()** 함수는 특정 소켓을 이 보안 세션과 연관시킵니다.
7. **gsk_secure_soc_init()** 함수는 SSL 환경 및 보안 세션에 속성 세트를 사용하여 비동기로 보안 세션 협상을 시작합니다.
8. **gsk_secure_soc_write()** 함수는 보안 세션에서 작업자 스레드에 자료를 기록합니다.

   주: GSKit 서버 예의 경우 이 함수는 **gsk_secure_soc_startRecv()** 함수가 완료되는 작업자 스레드에 자료를 기록합니다. 비동기 예에서는 완료된 **gsk_secure_soc_startInit()**에 기록합니다.
9. **gsk_secure_soc_read()** 함수는 보안 세션을 사용하여 작업자 스레드로부터 메세지를 수신합니다.
10. **gsk_secure_soc_close()** 함수는 보안 세션을 종료합니다.
11. **gsk_environment_close()** 함수는 SSL 환경을 종료합니다.
12. **close()** 함수는 연결을 종료합니다.

```
/* GSK Client Program using Application Id           */

/* This program assumes that the application id is   */
/* already registered and a certificate has been     */
/* associated with the appliciation id               */
/*                                                   */
/* No parameters, some comments and many hardcoded   */
/* values to keep it short and simple                */

/* use following command to create bound program:     */
/* CRTBNDC PGM(MYLIB/GSKCLIENT)                      */
/*         SRCFILE(MYLIB/CSRC)                       */
/*         SRCMBR(GSKCLIENT)                         */
```

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <gskssl.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <errno.h>
#define TRUE             1
#define FALSE            0

void main(void)
{
  gsk_handle my_env_handle=NULL;    /* secure environment handle */
  gsk_handle my_session_handle=NULL;    /* secure session handle */

    struct sockaddr_in address;
  int buf_len, rc = 0, sd = -1;
  int amtWritten, amtRead;
    char buff1[1024];
    char buff2[1024];

  /*  hardcoded IP address (change to make address were server program runs */
  char addr[16] = "1.1.1.1";

  /*******************************************/
  /* Issue all of the command in a do/while    */
  /* loop so that clean up can happen at end   */
  /*******************************************/
   do
  {
    /* open a gsk environment */
    rc = errno = 0;
    rc = gsk_environment_open(&my_env_handle);
  if (rc != GSK_OK)
    {
      printf("gsk_environment_open() failed with rc = %d and errno = %d.\n",
             rc,errno);
        printf("rc of %d means %s\n", rc, gsk_strerror(rc));
         break;
    }

    /* set the Application ID to use */
    rc = errno = 0;
    rc = gsk_attribute_set_buffer(my_env_handle,
                                  GSK_OS400_APPLICATION_ID,
                                  "MY_CLIENT_APP",
                                  13);
  if (rc != GSK_OK)
    {
      printf("gsk_attribute_set_buffer() failed with rc = %d and errno = %d.\n",
             rc,errno);
        printf("rc of %d means %s\n", rc, gsk_strerror(rc));
         break;
    }

    /* set this side as the client (this is the default */
    rc = errno = 0;
    rc = gsk_attribute_set_enum(my_env_handle,
```

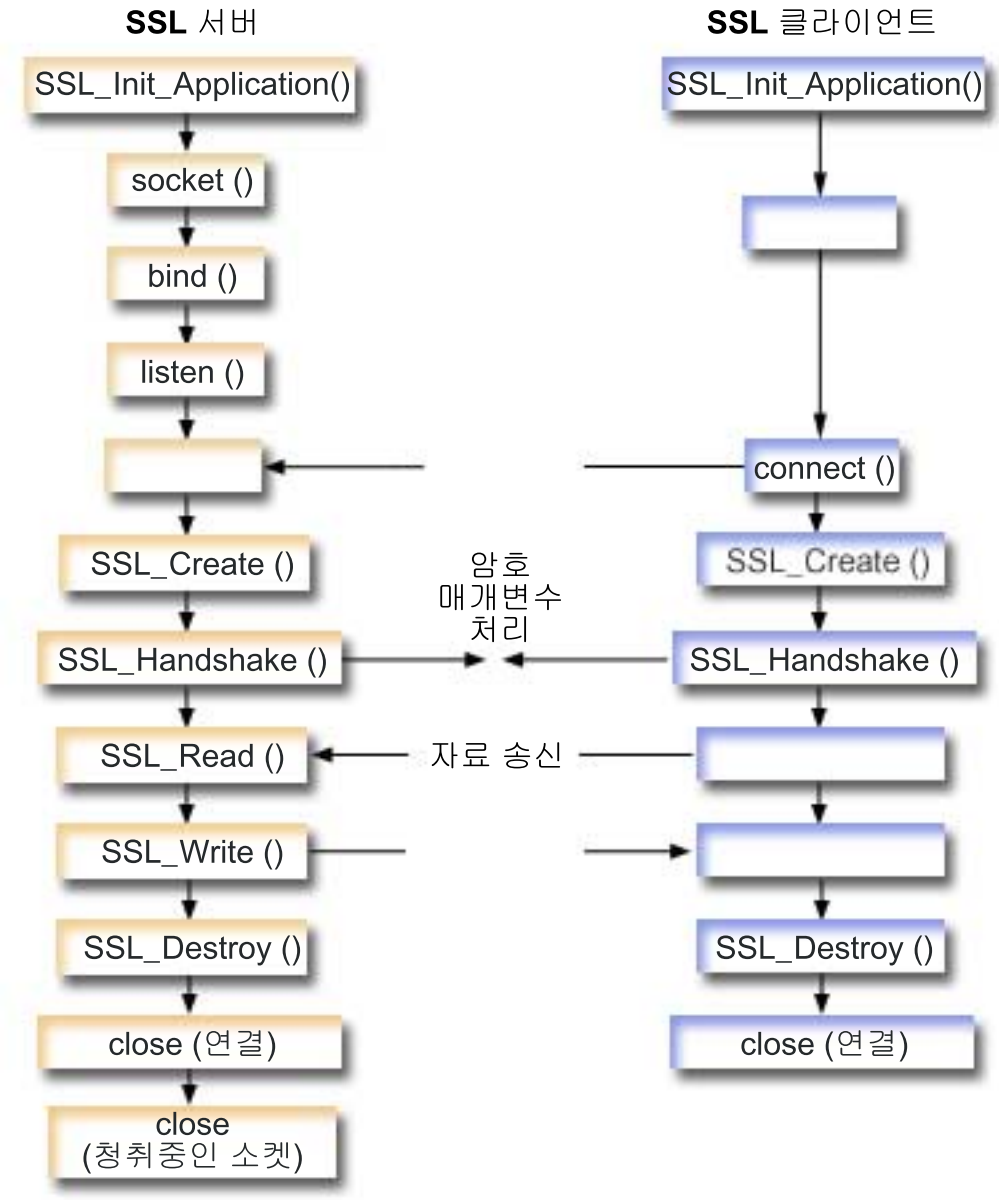
```
                                GSK_SESSION_TYPE,
                                GSK_CLIENT_SESSION);
  if (rc != GSK_OK)
    {
      printf("gsk_attribute_set_enum() failed with rc = %d and errno = %d.\n",
             rc,errno);
        printf("rc of %d means %s\n", rc, gsk_strerror(rc));
         break;
    }

    /* by default SSL_V2, SSL_V3, and TLS_V1 are enabled  */
    /* We will disable SSL_V2 for this example.           */
    rc = errno = 0;
    rc = gsk_attribute_set_enum(my_env_handle,
                                GSK_PROTOCOL_SSLV2,
                                GSK_PROTOCOL_SSLV2_OFF);
  if (rc != GSK_OK)
    {
      printf("gsk_attribute_set_enum() failed with rc = %d and errno = %d.\n",
             rc,errno);
        printf("rc of %d means %s\n", rc, gsk_strerror(rc));
         break;
    }

    /* set the cipher suite to use.  By default our default list     */
    /* of ciphers is enabled.  For this example we will just use one */
    rc = errno = 0;
    rc = gsk_attribute_set_buffer(my_env_handle,
                                  GSK_V3_CIPHER_SPECS,
                                  "05", /* SSL_RSA_WITH_RC4_128_SHA */
                                  2);
  if (rc != GSK_OK)
    {
      printf("gsk_attribute_set_buffer() failed with rc = %d and errno = %d.\n",
             rc,errno);
        printf("rc of %d means %s\n", rc, gsk_strerror(rc));
         break;
    }

    /* Initialize the secure environment */
    rc = errno = 0;
    rc = gsk_environment_init(my_env_handle);
  if (rc != GSK_OK)
    {
      printf("gsk_environment_init() failed with rc = %d and errno = %d.\n",
             rc,errno);
        printf("rc of %d means %s\n", rc, gsk_strerror(rc));
         break;
    }

    /* initialize a socket to be used for listening */
  sd = socket(AF_INET, SOCK_STREAM, 0);
    if (sd < 0)
    {
    perror("socket() failed");
         break;
    }
```

```
    /* connect to the server using a set port number */
    memset((char *) &address, 0, sizeof(address));
    address.sin_family = AF_INET;
    address.sin_port = 13333;
    address.sin_addr.s_addr = inet_addr(addr);
    rc = connect(sd, (struct sockaddr *) &address, sizeof(address));
      if (rc < 0)
    {
    perror("connect() failed");
         break;
    }

    /* open a secure session */
    rc = errno = 0;
    rc = gsk_secure_soc_open(my_env_handle, &my_session_handle);
  if (rc != GSK_OK)
    {
      printf("gsk_secure_soc_open() failed with rc = %d and errno = %d.\n",
             rc,errno);
        printf("rc of %d means %s\n", rc, gsk_strerror(rc));
         break;
    }

    /* associate our socket with the secure session */
    rc=errno=0;
    rc = gsk_attribute_set_numeric_value(my_session_handle,
                                         GSK_FD,
                                         sd);
  if (rc != GSK_OK)
    {
      printf("gsk_attribute_set_numeric_value() failed with rc = %d ", rc);
      printf("and errno = %d.\n", errno);
        printf("rc of %d means %s\n", rc, gsk_strerror(rc));
         break;
    }

    /* initiate the SSL handshake */
    rc = errno = 0;
    rc = gsk_secure_soc_init(my_session_handle);
  if (rc != GSK_OK)
    {
      printf("gsk_secure_soc_init() failed with rc = %d and errno = %d.\n",
             rc,errno);
        printf("rc of %d means %s\n", rc, gsk_strerror(rc));
         break;
    }

  /* memset buffer to hex zeros                    */
    memset((char *) buff1, 0, sizeof(buff1));

    /* send a message to the server using the secure session */
    strcpy(buff1,"Test of gsk_secure_soc_write \n\n");

    /* send the message to the client using the secure session */
    buf_len = strlen(buff1);
    amtWritten = 0;
    rc = gsk_secure_soc_write(my_session_handle, buff1, buf_len, &amtWritten);
    if (amtWritten != buf_len)
```

```
|      {
|    if (rc != GSK_OK)
|        {
|          printf("gsk_secure_soc_write() rc = %d and errno = %d.\n",rc,errno);
|          printf("rc of %d means %s\n", rc, gsk_strerror(rc));
|           break;
|        }
|     else
|        {
|          printf("gsk_secure_soc_write() did not write all data.\n");
|           break;
|        }
|      }
|
|      /* write results to screen */
|      printf("gsk_secure_soc_write() wrote %d bytes...\n", amtWritten);
|      printf("%s\n",buff1);
|
|    /* memset buffer to hex zeros                    */
|      memset((char *) buff2, 0x00, sizeof(buff2));
|
|      /* receive a message from the client using the secure session */
|      amtRead = 0;
|      rc = gsk_secure_soc_read(my_session_handle, buff2, sizeof(buff2), &amtRead);
|
|    if (rc != GSK_OK)
|      {
|        printf("gsk_secure_soc_read() rc = %d and errno = %d.\n",rc,errno);
|          printf("rc of %d means %s\n", rc, gsk_strerror(rc));
|           break;
|      }
|
|      /* write results to screen */
|      printf("gsk_secure_soc_read() received %d bytes, here they are ...\n",
|             amtRead);
|      printf("%s\n",buff2);
|
|
|    } while(FALSE);
|
|      /* disable SSL support for the socket  */
|    if (my_session_handle != NULL)
|      gsk_secure_soc_close(&my_session_handle);
|
|    /* disable the SSL environment */
|    if (my_env_handle != NULL)
|      gsk_environment_close(&my_env_handle);
|
|      /* close the connection */
|    if (sd > -1)
|              close(sd);
|
|    return;
|  }
```

예: SSL_ API를 사용하여 보안 서버 설정

GSKit API를 사용하여 보안 어플리케이션을 작성하는 것 외에 SSL_ API도 사용할 수 있습니다. 이 API는 iSeries 오퍼레이팅 시스템에 고유합니다. GSKit API의 경우와 마찬가지로 서버는 유효한 인증을 제공하여 자료를 안전하게 교환해야 합니다. 다음 그래픽은 보안 서버를 작성하는 데 사용되는 소켓 및 SSL_ API를 나타냅니다. IBM @server 플랫폼에 걸쳐 보안 어플리케이션을 작성하고 있는 경우 GSKit API를 사용하십시오.



이벤트의 소켓 흐름: SSL_ API를 사용하는 보안 서버

다음 설명은 SSL 서버가 SSL 클라이언트와 통신할 수 있도록 하는 API간 관계를 나타냅니다.

1. SSL 처리의 작업 환경을 초기화하고 현재 작업에서 실행될 모든 SSL 세션의 SSL 보안 정보를 설정하기 위한 **SSL_Init()** 또는 **SSL_Init_Application()** 호출. 이러한 API 중 하나만이 사용되어야 합니다. **SSL_Init_Application()** API를 사용하는 것이 더 좋습니다.

   주: 다음 프로그램 예에서는 **SSL_Init_Application** API를 사용합니다.
2. 서버는 **socket()**을 호출하여 소켓 설명자를 확보합니다.
3. 서버는 **bind()**, **listen()** 및 **accept()**를 호출하여 서버 프로그램에 대한 연결을 활성화합니다.
4. 서버는 **SSL_Create()**를 호출하여 연결된 소켓에 대해 SSL 지원이 가능하도록 합니다.
5. 서버는 **SSL_Handshake()**를 호출하여 암호화 매개변수의 SSL 핸드쉐이크 조정을 초기화합니다.
6. 서버는 **SSL_Write()** 및 **SSL_Read()**를 호출하여 자료를 송/수신합니다.
7. 서버는 **SSL_Destroy()**를 호출하여 소켓에 대한 SSL 지원을 사용할 수 없게 합니다.
8. 서버는 **close()**를 호출하여 연결된 소켓을 소멸시킵니다.

이벤트의 소켓 흐름: SSL_ API를 사용하는 보안 클라이언트

1. SSL 처리의 작업 환경을 초기화하고 현재 작업에서 실행될 모든 SSL 세션의 SSL 보안 정보를 설정하기 위한 **SSL_Init()** 또는 **SSL_Init_Application()** 호출. 이러한 API 중 하나만이 사용되어야 합니다. **SSL_Init_Application** API를 사용하는 것이 더 좋습니다.

   주: 다음 프로그램 예에서는 **SSL_Init_Application** API를 사용합니다.
2. 클라이언트는 **socket()**을 호출하여 소켓 설명자를 확보합니다.
3. 클라이언트는 **connect()**를 호출하여 클라이언트 프로그램에 대한 연결을 활성화합니다.
4. 클라이언트는 **SSL_Create()**를 호출하여 연결된 소켓에 대해 SSL 지원이 가능하도록 합니다.
5. 클라이언트는 **SSL_Handshake()**를 호출하여 암호화 매개변수의 SSL 핸드쉐이크 조정을 초기화합니다.
6. 클라이언트는 **SSL_Read()** 및 **SSL_Write()**를 호출하여 자료를 송수신합니다.
7. 클라이언트는 **SSL_Destroy()**를 호출하여 소켓에 대한 SSL 지원을 사용할 수 없게 합니다.
8. 클라이언트는 **close()**를 호출하여 연결된 소켓을 소멸시킵니다.

주: 샘플에서는 AF_INET 주소 패밀리를 사용하지만 AF_INET6 주소 패밀리를 사용하도록 수정할 수 있습니다.

코드 예를 사용하는 것에 대한 정보는 코드 면책사항을 참조하십시오.

```
/* SSL Server Program using SSL_Init_Application   */

/* Assummes that application id is already registered */
/* and a certificate has been associated with the     */
/* application id.                                    */
/* No parameters, some comments and many hardcoded    */
/* values to keep it short and simple                 */

/* use following command to create bound program:      */
/* CRTBNDC PGM(MYLIB/SSLSERVAPP)                      */
/*         SRCFILE(MYLIB/CSRC)                        */
/*         SRCMBR(SSLSERVAPP)                         */
```

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <ssl.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <errno.h>

void main(void)
{
  SSLHandle *sslh;
    SSLInitApp sslinit;

    struct sockaddr_in address;
    int buf_len, on = 1, rc = 0, sd, lsd, al;
    char buff[1024];

    /*  only want to use 1 cipher suite     */
    unsigned short int cipher = SSL_RSA_WITH_RC4_128_SHA;

    void * malloc_ptr = (void *) NULL;
    unsigned int malloc_size = 8192;

    /* memset sslinitapp structure to hex zeros */
    memset((char *)&sslinit, 0, sizeof(sslinit));

    /* fill in values for sslinit app structure */
    sslinit.applicationID = "MY_SERVER_APP";
    sslinit.applicationIDLen = 13;
    sslinit.localCertificate = NULL;
    sslinit.localCertificateLen = 0;
    sslinit.cipherSuiteList = NULL;
    sslinit.cipherSuiteListLen = 0;

    /* allocate and set pointers for certificate buffer */
    malloc_ptr = (void*) malloc(malloc_size);
    sslinit.localCertificate = (unsigned char*) malloc_ptr;
    sslinit.localCertificateLen = malloc_size;

    /* initialize ssl call SSL_Init_Application */
    rc = SSL_Init_Application(&sslinit);
  if (rc != 0)
    {
      printf("SSL_Init_Application() failed with rc = %d and errno = %d.\n",
             rc,errno);
  return;
    }

    /* initialize a socket to be used for listening */
    lsd = socket(AF_INET, SOCK_STREAM, 0);
    if (lsd < 0)
    {
    perror("socket() failed");
  return;
    }
```

```
  /* set socket so can be reused immediately */
  rc = setsockopt(lsd, SOL_SOCKET,
                  SO_REUSEADDR,
                   (char *)&on,
                 sizeof(on));
    if (rc < 0)
  {
  perror("setsockopt() failed");
return;
  }

  /* bind to the local server address */
  memset((char *) &address, 0, sizeof(address));
  address.sin_family = AF_INET;
  address.sin_port = 13333;
  address.sin_addr.s_addr = 0;
  rc = bind(lsd, (struct sockaddr *) &address, sizeof(address));
    if (rc < 0)
  {
  perror("bind() failed");
  close(lsd);
return;
  }

  /* enable the socket for incoming client connections */
  listen(lsd, 5);
    if (rc < 0)
  {
  perror("listen() failed");
  close(lsd);
return;
  }

  /* accept an incoming client connection */
  al = sizeof(address);
  sd = accept(lsd, (struct sockaddr *) &address, &al);
  if (sd < 0)
  {
    perror("accept() failed");
  close(lsd);
return;
  }

  /* enable SSL support for the socket */
sslh = SSL_Create(sd, SSL_ENCRYPT);
if (sslh == NULL)
  {
    printf("SSL_Create() failed with errno = %d.\n", errno);
  close(lsd);
         close(sd);
return;
  }

  /* set up parameters for handshake */
  sslh -> protocol = 0;
sslh -> timeout = 0;
  sslh -> cipherSuiteList = &cipher;
  sslh -> cipherSuiteListLen = 1;
```

```
  /* initiate the SSL handshake */
  rc = SSL_Handshake(sslh, SSL_HANDSHAKE_AS_SERVER);
if (rc != 0)
  {
    printf("SSL_Handshake() failed with rc = %d and errno = %d.\n",
           rc,errno);
SSL_Destroy(sslh);
  close(lsd);
          close(sd);
return;
  }

/* memset buffer to hex zeros                    */
  memset((char *) buff, 0, sizeof(buff));

  /* receive a message from the client using the secure session */
  rc = SSL_Read(sslh, buff, sizeof(buff));
    if (rc < 0)
  {
    printf("SSL_Read() rc = %d and errno = %d.\n",rc,errno);
    rc = SSL_Destroy(sslh);
if (rc != 0)
      printf("SSL_Destroy() rc = %d and errno = %d.\n",rc,errno);
  close(lsd);
          close(sd);
return;
  }

  /* write results to screen */
  printf("SSL_Read() read ...\n");
  printf("%s\n",buff);

  /* send the message to the client using the secure session */
  buf_len = strlen(buff);
  rc = SSL_Write(sslh, buff, buf_len);
  if (rc != buf_len)
  {
    if (rc < 0)
    {
    printf("SSL_Write() failed with rc = %d.\n",rc);
SSL_Destroy(sslh);
  close(lsd);
          close(sd);
return;
    }
 else
    {
    printf("SSL_Write() did not write all data.\n");
SSL_Destroy(sslh);
  close(lsd);
          close(sd);
return;
    }
  }

  /* write results to screen */
  printf("SSL_Write() wrote ...\n");
```

```
    printf("%s\n",buff);

    /* disable SSL support for the socket  */
  SSL_Destroy(sslh);

    /* close the connection */
            close(sd);

    /* close the listening socket */
    close(lsd);

  return;
}
```

예: SSL_ API를 사용하여 보안 클라이언트 설정

GSKit API 이외에 OS/400 소켓은 일반적인 SSL_ API도 지원합니다. 이 API는 iSeries 고유 서버와 클라이언트 어플리케이션간에 보안 연결을 설정합니다. 이 프로그램 및 해당 서버 어플리케이션에 대한 이벤트의 소켓 흐름을 설명하는 그래픽은 예: SSL_ API를 사용하여 보안 서버 설정을 참조하십시오. 다음은 SSL_ API를 사용하는 클라이언트 어플리케이션이 SSL_API를 사용하는 서버 어플리케이션과 통신할 수 있도록 하는 예입니다.

코드 예를 사용하는 것에 대한 정보는 코드 면책사항을 참조하십시오.

```
/* SSL Client Program using SSL_Init_Application   */

/* Assummes that application id is already registered */
/* and a certificate has been associated with the     */
/* application id.                                    */
/* No parameters, some comments and many hardcoded    */
/* values to keep it short and simple                 */

/* use following command to create bound program:      */
/* CRTBNDC PGM(MYLIB/SSLCLIAPP)                       */
/*         SRCFILE(MYLIB/CSRC)                        */
/*         SRCMBR(SSLCLIAPP                            */

#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <ctype.h>
#include <sys/socket.h>
#include <ssl.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>
#include <errno.h>

/*  Making this simple - no parameters  */
void main(void)
{
  SSLHandle *sslh;
    SSLInitApp sslinit;
    struct sockaddr_in address;
    int buf_len, rc = 0, sd;
    char buff1[1024];
```

```
  char buff2[1024];

  /*  only want to use 1 cipher suite     */
  unsigned short int cipher = SSL_RSA_WITH_RC4_128_SHA;

  /*  hardcoded IP address                */
  char addr[12] = "16.35.146.84";

  void * malloc_ptr = (void *) NULL;
  unsigned int malloc_size = 8192;

  /* memset sslinit structure to hex zeros */
  memset((char *)&sslinit, 0, sizeof(sslinit));

  /* fill in values for sslinitapp structure */
  /* using an existing app id                      */
  sslinit.applicationID = "MY_CLIENT_APP";
  sslinit.applicationIDLen = 13;
  sslinit.localCertificate = NULL;
  sslinit.localCertificateLen = 0;
  sslinit.cipherSuiteList = NULL;
  sslinit.cipherSuiteListLen = 0;

  /* allocate and set pointers for certificate buffer */
  malloc_ptr = (void*) malloc(malloc_size);
  sslinit.localCertificate = (unsigned char*) malloc_ptr;
  sslinit.localCertificateLen = malloc_size;

  /* initialize ssl call SSL_Init_Application */
  rc = SSL_Init_Application(&sslinit);
if (rc != 0)
  {
    printf("SSL_Init_Application() failed with rc = %d and errno = %d.\n",
           rc,errno);
return;
  }

  /* initialize a socket */
sd = socket(AF_INET, SOCK_STREAM, 0);
  if (sd < 0)
  {
  perror("socket() failed");
return;
  }

  /* enable SSL support for the socket */
sslh = SSL_Create(sd, SSL_ENCRYPT);
if (sslh == NULL)
  {
    printf("SSL_Create() failed with errno = %d.\n", errno);
         close(sd);
return;
  }

  /* connect to the server using a set port number */
  memset((char *) &address, 0, sizeof(address));
  address.sin_family = AF_INET;
  address.sin_port = 13333;
```

```
  address.sin_addr.s_addr = inet_addr(addr);
  rc = connect(sd, (struct sockaddr *) &address, sizeof(address));
    if (rc < 0)
  {
  perror("connect() failed");
          close(sd);
return;
  }

  /* set up to call handshake, setting cipher     */
  sslh -> protocol = 0;
sslh -> timeout = 0;
  sslh -> cipherSuiteList = &cipher;
  sslh -> cipherSuiteListLen = 1;

  /* initiate the SSL handshake - as a CLIENT */
rc = SSL_Handshake(sslh, SSL_HANDSHAKE_AS_CLIENT);
if (rc != 0)
  {
    printf("SSL_Handshake() failed with rc = %d and errno = %d.\n",
           rc,errno);
          close(sd);
return;
  }

  /* send a message to the server using the secure session */
  strcpy(buff1,"Test of SSL_Write \n\n");
  buf_len = strlen(buff1);
  rc = SSL_Write(sslh, buff1, buf_len);
  if (rc != buf_len)
  {
    if (rc < 0)
    {
     printf("SSL_Write() failed with rc = %d and errno = %d.\n",rc,
             errno);
SSL_Destroy(sslh);
          close(sd);
return;
    }
 else
    {
    printf("SSL_Write() did not write all data.\n");
SSL_Destroy(sslh);
          close(sd);
return;
    }
  }

  /* write the results to the screen */
  printf("SSL_Write() wrote ...\n");
  printf("%s\n",buff1);

  memset((char *) buff2, 0x00, sizeof(buff2));

  /* receive the message from the server using the secure session */
  rc = SSL_Read(sslh, buff2, buf_len);
    if (rc < 0)
  {
```

```
    printf("SSL_Read() failed with rc = %d.\n",rc);
  SSL_Destroy(sslh);
            close(sd);
  return;
    }

    /* write the results to the screen */
    printf("SSL_Read() read ...\n");
    printf("%s\n",buff2);

    /* disable SSL support for the socket  */
  SSL_Destroy(sslh);

    /* close the connection by closing the local socket */
            close(sd);
  return;
}
```

예: 스레드세이프 네트워크 루틴에 gethostbyaddr_r() 사용

다음은 **gethostbyaddr_r()**를 사용하는 프로그램의 한 예입니다. 이름이 ″_r″로 끝나는 다른 모든 루틴은 유사한 의미론을 가지고 있으므로 스레드세이프 상태를 수반합니다. 이 예 프로그램은 점 십진 표기법으로 IP 주소를 사용하고 호스트명을 인쇄합니다.

코드 예를 사용하는 것에 대한 정보는 코드 면책사항을 참조하십시오.

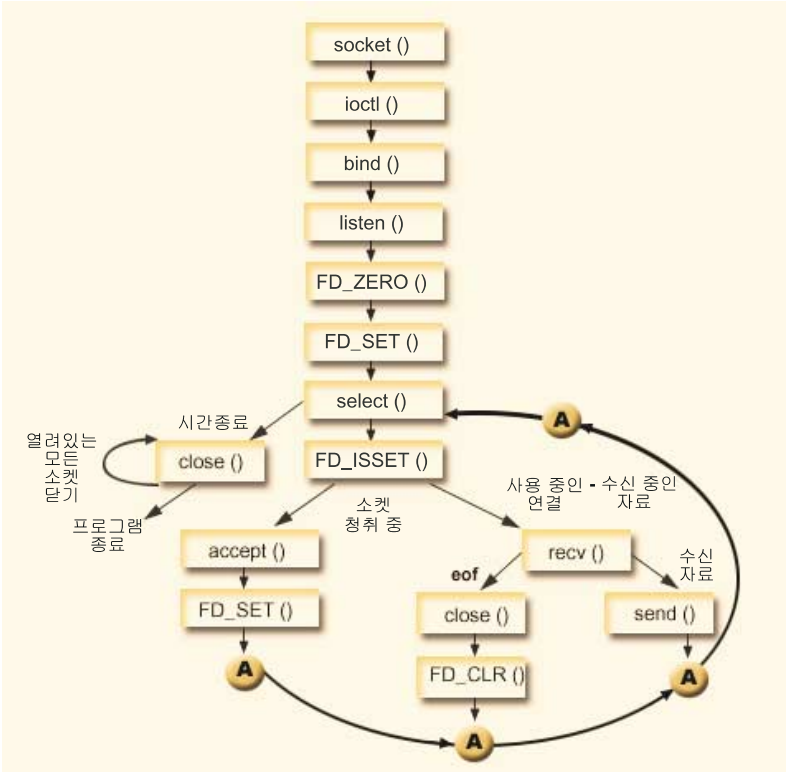
```
/*********************************************************/
/* Header files                                          */
/*********************************************************/
#include </netdb.h>
#include <sys/param.h>
#include <netinet/in.h>
#include <stdlib.h>
#include <stdio.h>
#include <arpa/inet.h>
#include <sys/socket.h>
#define HEX00 '\x00'
#define NUMPARMS 2
/*********************************************************/
/* Pass one parameter that is the IP address in          */
/* dotted decimal notation. The host name will be        */
/* displayed if found; otherwise, a message states       */
/* host not found.                                       */
/*********************************************************/
int main (int argc, char *argv[])
    {
    int rc;
    struct in_addr internet_address;
    struct hostent hst_ent;
    struct hostent_data  hst_ent_data;
    char dotted_decimal_address [16];
    char host_name[MAXHOSTNAMELEN];

    /*********************************************************/
    /* Verify correct number of arguments have been passed    */
```

```
    /*********************************************************/
    if (argc != NUMPARMS)
     {
      printf("Wrong number of parms passed\n");
          exit(-1);
     }
    /*********************************************************/
    /* Obtain addressability to parameters passed            */
    /*********************************************************/
    strcpy(dotted_decimal_address, argv[1]);

    /*********************************************************/
    /* Initialize the structure-field                        */
    /* hostent_data.host_control_blk with hexadecimal zeros  */
    /* before its initial use. If you require compatibility  */
    /* with other platforms, then you must initialize the    */
    /* entire hostent_data structure with hexadecimal zeros. */
    /*********************************************************/
    /* Initialize to hex 00 hostent_data structure           */
    /*********************************************************/
    memset(&hst_ent_data,HEX00,sizeof(struct hostent_data));

    /*********************************************************/
    /* Translate an Internet address from dotted decimal     */
    /* notation to 32-bit IP address format.                 */
    /*********************************************************/
    internet_address.s_addr=inet_addr(dotted_decimal_address);

    /*********************************************************/
    /* Obtain host name                                      */
    /*********************************************************/
    /*********************************************************/
    /* NOTE: The gethostbyaddr_r() returns an integer.       */
    /* The following are possible values:                    */
    /*   -1 (unsuccessful call)                              */
    /*   0 (successful call)                                 */
    /*********************************************************/
    rc=gethostbyaddr_r((char *) &internet_address,
                          sizeof(struct in_addr), AF_INET,
                       &hst_ent, &hst_ent_data);

if (rc== -1)
{
         printf("Host name not found\n");
          exit(-1);
       }
   else
       {
        /*****************************************************/
        /*   Copy the host name to an output buffer          */
        /*****************************************************/
          (void) memcpy((void *) host_name,
        /*****************************************************/
        /* You must address all the results through the     */
        /* hostent structure hst_ent.                        */
        /* NOTE: Hostent_data structure hst_ent_data is just */
        /* a data repository that is used to support the     */
        /* hostent structure. Applications should consider  */
```

```
    /* hostent_data a storage area to put host level data */
    /* that the application does not need to access.    */
    /*****************************************************/
                   (void *) hst_ent.h_name,
                   MAXHOSTNAMELEN);
    /*****************************************************/
    /* Print the host name                               */
    /*****************************************************/
      printf("The host name is %s\n", host_name);

     }
  exit(0);
}
```

예: 비블록화 I/O 및 select()

다음 샘플 프로그램에서는 비블록화 및 select() API를 사용합니다. 이 예에 사용할 수 있는 공통 클라이언트 작업의 코드가 들어 있는 예를 보려면, 예: 총칭 클라이언트를 참조하십시오.



이벤트의 소켓 흐름: 비블록화 I/O 및 select()를 사용하는 서버

예에서 사용되는 호출은 다음과 같습니다.

1. **socket()** 함수는 종료점을 표시하는 소켓 설명자를 리턴합니다. 이 명령문은 TCP 전송(SOCK_STREAM)을 사용하는 INET(인터넷 프로토콜) 주소 패밀리가 이 소켓에 사용된다는 것도 식별합니다.
2. **ioctl()** 함수는 필수 대기 시간이 만기되기 전에 서버가 재시작될 때 로컬 주소를 재사용할 수 있도록 합니다. 이 예에서는 소켓을 비블록화로 설정합니다. 청취 소켓에서 그 상태를 계승하기 때문에 수신 연결을 위한 모든 소켓도 비블록화됩니다.

3. 소켓 설명자가 작성된 후에 **bind()** 함수는 소켓에 고유한 이름을 제공합니다.
4. **listen()**은 서버가 수신 클라이언트 연결을 허용할 수 있도록 합니다.
5. 서버에서는 수신 연결 요구를 승인하기 위해 **accept()** 함수를 사용합니다. **accept()** 호출은 수신 연결이 도착하길 기다리며 무기한 블록화됩니다.
6. **select()** 함수는 프로세스가 이벤트 발생을 기다릴 수 있도록 하며 이벤트가 발생할 때 프로세스를 깨울 수 있도록 합니다. 이 예에서 **select ()** 함수는 처리될 준비가 된 소켓 설명자를 나타내는 숫자를 리턴합니다.

   0 프로세스가 시간종료될 것임을 나타냅니다. 이 예에서 시간종료는 30초로 설정됩니다.

   -1 프로세스가 실패했음을 나타냅니다.

   1 하나의 설명자만 처리될 준비가 되었음을 나타냅니다. 이 예에서는 1이 리턴될 때 FD_ISSET 및 후속적인 소켓 호출은 한 번만 완료됩니다.

   n 처리를 기다리는 설명자가 여러 개임을 나타냅니다. 이 예에서는 n이 리턴될 때 FD_ISSET 및 후속적인 코드가 루프되어 서버가 요구를 수신하는 순서대로 요구를 완료합니다.
7. **accept()** 및 **recv()** 함수는 EWOULDBLOCK이 리턴될 때 완료됩니다.
8. **send()** 함수는 클라이언트로 자료를 다시 에코우합니다.
9. **close()** 함수는 열린 소켓 설명자를 닫습니다.

코드 예를 사용하는 것에 대한 정보는 코드 면책사항을 참조하십시오.

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/ioctl.h>
#include <sys/socket.h>
#include <sys/time.h>
#include <netinet/in.h>
#include <errno.h>

#define SERVER_PORT 12345

#define TRUE             1
#define FALSE            0

main (int argc, char *argv[])
{
   int    i, len, rc, on = 1;
   int    listen_sd, max_sd, new_sd;
   int    desc_ready, end_server = FALSE;
   int    close_conn;
   char   buffer[80];
   struct sockaddr_in   addr;
  struct timeval timeout;
   struct fd_set        master_set, working_set;

   /*************************************************************/
   /* Create an AF_INET stream socket to receive incoming       */
   /* connections on                                            */
   /*************************************************************/
   listen_sd = socket(AF_INET, SOCK_STREAM, 0);
```

```
 if (listen_sd < 0)
 {
  perror("socket() failed");
        exit(-1);
 }

 /*************************************************************/
/* Allow socket descriptor to be reuseable */
 /*************************************************************/
 rc = setsockopt(listen_sd, SOL_SOCKET,  SO_REUSEADDR,
                 (char *)&on, sizeof(on));
    if (rc < 0)
 {
  perror("setsockopt() failed");
          close(listen_sd);
        exit(-1);
 }

 /*************************************************************/
 /* Set socket to be non-blocking.  All of the sockets for    */
 /* the incoming connections will also be non-blocking since  */
 /* they will inherit that state from the listening socket.   */
 /*************************************************************/
 rc = ioctl(listen_sd, FIONBIO, (char *)&on);
    if (rc < 0)
 {
perror("ioctl() failed");
          close(listen_sd);
        exit(-1);
 }

 /*************************************************************/
 /* Bind the socket                                           */
 /*************************************************************/
 memset(&addr, 0, sizeof(addr));
 addr.sin_family = AF_INET;
    addr.sin_addr.s_addr = htonl(INADDR_ANY);
 addr.sin_port   = htons(SERVER_PORT);
 rc = bind(listen_sd,
           (struct sockaddr *)&addr, sizeof(addr));
    if (rc < 0)
 {
  perror("bind() failed");
          close(listen_sd);
        exit(-1);
 }

 /*************************************************************/
 /* Set the listen back log                                   */
 /*************************************************************/
 rc = listen(listen_sd, 32);
    if (rc < 0)
 {
  perror("listen() failed");
          close(listen_sd);
        exit(-1);
 }
```

```
 /*************************************************************/
 /* Initialize the master fd_set                              */
 /*************************************************************/
 FD_ZERO(&master_set);
 max_sd = listen_sd;
 FD_SET(listen_sd, &master_set);

 /*************************************************************/
 /* Initialize the timeval struct to 3 minutes.  If no        */
 /* activity after 3 minutes this program will end.           */
 /*************************************************************/
 timeout.tv_sec  = 3 * 60;
timeout.tv_usec = 0;

 /*************************************************************/
 /* Loop waiting for incoming connects or for incoming data   */
 /* on any of the connected sockets.                          */
 /*************************************************************/
 do
 {
    /**********************************************************/
    /* Copy the master fd_set over to the working fd_set.     */
    /**********************************************************/
    memcpy(&working_set, &master_set, sizeof(master_set));

    /**********************************************************/
    /* Call select() and wait 5 minutes for it to complete.   */
    /**********************************************************/
    printf("Waiting on select()...\n");
    rc = select(max_sd + 1, &working_set, NULL, NULL, &timeout);

    /**********************************************************/
    /* Check to see if the select call failed.                */
    /**********************************************************/
    if (rc < 0)
    {
  perror("select() failed");
        break;
    }

    /**********************************************************/
    /* Check to see if the 5 minute time out expired.         */
    /**********************************************************/
    if (rc == 0)
    {
       printf("  select() timed out.  End program.\n");
       break;
    }

    /**********************************************************/
    /* One or more descriptors are readable.  Need to         */
    /* determine which ones they are.                         */
    /**********************************************************/
    desc_ready = rc;
    for (i=0; i <= max_sd  &&  desc_ready > 0; ++i)
    {
       /*******************************************************/
       /* Check to see if this descriptor is ready            */
```

```
      /*************************************************/
      if (FD_ISSET(i, &working_set))
      {
         /************************************************/
         /* A descriptor was found that was readable - one   */
         /* less has to be looked for.  This is being done   */
         /* so that we can stop looking at the working set   */
         /* once we have found all of the descriptors that   */
         /* were ready.                                      */
         /************************************************/
         desc_ready -= 1;

         /************************************************/
         /* Check to see if this is the listening socket     */
         /************************************************/
         if (i == listen_sd)
         {
            printf("  Listening socket is readable\n");
            /*********************************************/
            /* Accept all incoming connections that are      */
            /* queued up on the listening socket before we   */
            /* loop back and call select again.              */
            /*********************************************/
do
            {
               /******************************************/
               /* Accept each incoming connection.  If       */
               /* accept fails with EWOULDBLOCK, then we     */
               /* have accepted all of them.  Any other      */
               /* failure on accept will cause us to end the */
               /* server.                                    */
               /******************************************/
               new_sd = accept(listen_sd, NULL, NULL);
               if (new_sd < 0)
               {
                  if (errno != EWOULDBLOCK)
                  {
   perror("accept() failed");
                     end_server = TRUE;
                  }
      break;
               }

               /******************************************/
               /* Add the new incoming connection to the     */
               /* master read set                            */
               /******************************************/
               printf("  New incoming connection - %d\n", new_sd);
               FD_SET(new_sd, &master_set);
               if (new_sd > max_sd)
                  max_sd = new_sd;

               /******************************************/
               /* Loop back up and accept another incoming   */
               /* connection                                 */
               /******************************************/
            } while (new_sd != -1);
         }
```

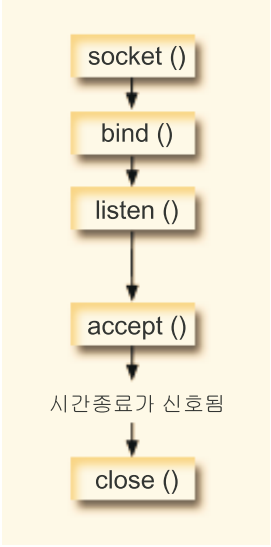
```
         /*************************************************/
         /* This is not the listening socket, therefore an   */
         /* existing connection must be readable             */
         /*************************************************/
else
         {
            printf("  Descriptor %d is readable\n", i);
            close_conn = FALSE;
            /*********************************************/
            /* Receive all incoming data on this socket      */
            /* before we loop back and call select again.    */
            /*********************************************/
do
            {
               /*********************************************/
               /* Receive data on this connection until the  */
               /* recv fails with EWOULDBLOCK.  If any other */
               /* failure occurs, we will close the          */
               /* connection.                                */
               /*********************************************/
               rc = recv(i, buffer, sizeof(buffer), 0);
   if (rc < 0)
               {
                  if (errno != EWOULDBLOCK)
                  {
   perror("recv() failed");
                  close_conn = TRUE;
                  }
      break;
               }

               /*********************************************/
               /* Check to see if the connection has been    */
               /* closed by the client                       */
               /*********************************************/
   if (rc == 0)
               {
                  printf("  Connection closed\n");
                  close_conn = TRUE;
      break;
               }

               /*********************************************/
               /* Data was recevied                          */
               /*********************************************/
len = rc;
printf("%d bytes received\n", len);

               /*********************************************/
/* Echo the data back to the client              */
               /*********************************************/
               rc = send(i, buffer, len, 0);
   if (rc < 0)
               {
   perror("send() failed");
                  close_conn = TRUE;
      break;
```

```
                  }

               } while (TRUE);

               /*************************************************/
               /* If the close_conn flag was turned on, we need */
               /* to clean up this active connection.  This     */
               /* clean up process includes removing the        */
               /* descriptor from the master set and            */
               /* determining the new maximum descriptor value  */
               /* based on the bits that are still turned on in */
               /* the master set.                               */
               /*************************************************/
               if (close_conn)
               {
          close(i);
                  FD_CLR(i, &master_set);
                  if (i == max_sd)
                  {
                     while (FD_ISSET(max_sd, &master_set) == FALSE)
                        max_sd -= 1;
                  }
               }
            } /* End of existing connection is readable */
         } /* End of if (FD_ISSET(i, &working_set)) */
      } /* End of loop through selectable descriptors */

   } while (end_server == FALSE);

   /*************************************************************/
   /* Cleanup all of the sockets that are open                  */
   /*************************************************************/
   for (i=0; i <= max_sd; ++i)
   {
      if (FD_ISSET(i, &master_set))
         close(i);
   }
}
```

예: 블록화 소켓 API에 신호 사용

신호를 사용하면 프로세스 또는 어플리케이션이 블록화될 때 통지를 받을 수 있습니다. 신호는 프로세스 블록화를 위한 시간 제한을 제공합니다. 이 예에서 신호는 **accept()** 호출시 5초 후에 발생합니다. 이 호출은 보통 무기한 블록화되지만 경보를 설정했으므로 5초 동안만 블록화됩니다. 블록화된 프로그램은 어플리케이션 또는 서버의 성능을 방해할 수 있기 때문에 이런 충격을 줄이기 위해 사용할 수 있습니다. 다음 예에서는 블록화 소켓 API에 신호를 사용하는 방법을 보여줍니다.

주: 스레드 서버 모델에 사용된 I/O는 보다 일반적인 모델보다 더 좋습니다. 비동기 I/O 사용 장점에 대한 자세한 정보는 비동기 I/O를 참조하십시오. 비동기 I/O API를 사용하는 샘플 프로그램은 예: 비동기I/O 사용을 참조하십시오.



이벤트의 소켓 흐름: 소켓 블록화로 신호 사용

다음 기능 호출 순서는 소켓이 비활동 상태일 때 신호를 사용하여 어플리케이션에 경고하는 방법을 나타냅니다.

1. **socket()** 함수는 종료점을 표시하는 소켓 설명자를 리턴합니다. 이 명령문은 UDP 전송(SOCK_DGRAM)을 사용하는 INET(인터넷 프로토콜) 주소 패밀리가 이 소켓에 사용된다는 것도 식별합니다.
2. 소켓 설명자가 작성된 후에 **bind()** 함수는 소켓에 고유한 이름을 제공합니다. 이 예에서는 클라이언트 어플리케이션이 이 소켓에 연결하지 않기 때문에 포트 번호가 지정되지 않습니다. **accept()** 같은 블록화 API를 사용하는 다른 서버 프로그램 내에서 이 코드 스니펫을 사용할 수 있습니다.
3. **listen()** 함수는 클라이언트 연결 요청을 승인할 것임을 표시합니다. **listen()** 함수가 발행되고 나서 5초 후에 경보가 꺼지도록 설정됩니다. 이 경보 또는 신호는 **accept()** 호출이 블록화될 때 사용자에게 경고합니다.
4. **accept()** 함수는 클라이언트 연결 요구를 허용합니다. 이 호출은 보통 무기한 블록화되지만 경보를 설정했으므로 5초 동안만 블록화됩니다. 경보가 꺼질 때 accept 호출은 -1 및 EINTR errno 값으로 완료됩니다.
5. **close()** 함수는 열린 소켓 설명자를 종료합니다.

코드 예를 사용하는 것에 대한 정보는 코드 면책사항을 참조하십시오.

```
/*********************************************************************/
/* Example shows how to set alarms for blocking socket APIs          */
/*********************************************************************/

/*********************************************************************/
/* Include files                                                     */
/*********************************************************************/
#include <signal.h>
#include <unistd.h>
#include <stdio.h>
#include <time.h>
#include <errno.h>
#include <sys/socket.h>
```

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

/*********************************************************************/
/* Signal catcher routine.  This routine will be called when the  */
/* signal occurs.                                                    */
/*********************************************************************/
void catcher(int sig)
{
   printf("   Signal catcher called for signal %d\n", sig);
}

/*********************************************************************/
/* Main program                                                      */
/*********************************************************************/
int main (int argc, char *argv[])
{
   struct sigaction sact;
   struct sockaddr_in   addr;
   time_t t;
  int sd, rc;

/*********************************************************************/
/* Create an AF_INET, SOCK_STREAM socket                             */
/*********************************************************************/
   printf("Create a TCP socket\n");
  sd = socket(AF_INET, SOCK_STREAM, 0);
   if (sd == -1)
   {
      perror("   socket failed");
    return(-1);
   }

/*********************************************************************/
/* Bind the socket.  A port number was not specified because      */
/* we are not going to ever connect to this socket.               */
/*********************************************************************/
   memset(&addr, 0, sizeof(addr));
   addr.sin_family = AF_INET;
   printf("Bind the socket\n");
   rc = bind(sd, (struct sockaddr *)&addr, sizeof(addr));
  if (rc != 0)
   {
      perror("   bind failed");
            close(sd);
      return(-2);
   }

/*********************************************************************/
/* Perform a listen on the socket.                                   */
/*********************************************************************/
   printf("Set the listen backlog\n");
   rc = listen(sd, 5);
  if (rc != 0)
   {
      perror("   listen failed");
            close(sd);
      return(-3);
   }
```

```
/*****************************************************************/
/* Set up an alarm that will go off in 5 seconds.                */
/*****************************************************************/
   printf("\nSet an alarm to go off in 5 seconds.  This alarm will cause the\n");
   printf("blocked accept() to return a -1 and an errno value of EINTR.\n\n");
   sigemptyset(&sact.sa_mask);
   sact.sa_flags = 0;
   sact.sa_handler = catcher;
   sigaction(SIGALRM, &sact, NULL);
   alarm(5);

/*****************************************************************/
/* Display the current time when the alarm was set               */
/*****************************************************************/
   time(&t);
   printf("Before accept(), time is %s", ctime(&t));

/*****************************************************************/
/* Call accept.  This call will normally block indefinitely,     */
/* but because we have an alarm set, it will only block for      */
/* 5 seconds.  When the alarm goes off, the accept call will     */
/* complete with -1 and an errno value of EINTR.                 */
/*****************************************************************/
   errno = 0;
   printf("   Wait for an incoming connection to arrive\n");
   rc = accept(sd, NULL, NULL);
   printf("   accept() completed.  rc = %d, errno = %d\n", rc, errno);
   if (rc >= 0)
   {
      printf("   Incoming connection was recevied\n");
      close(rc);
   }
   else
   {
      perror("   errno string");
   }

/*****************************************************************/
/* Show what time it was when the alarm went off                 */
/*****************************************************************/
   time(&t);
   printf("After accept(), time is %s\n", ctime(&t));
            close(sd);
  return(0);
}
```

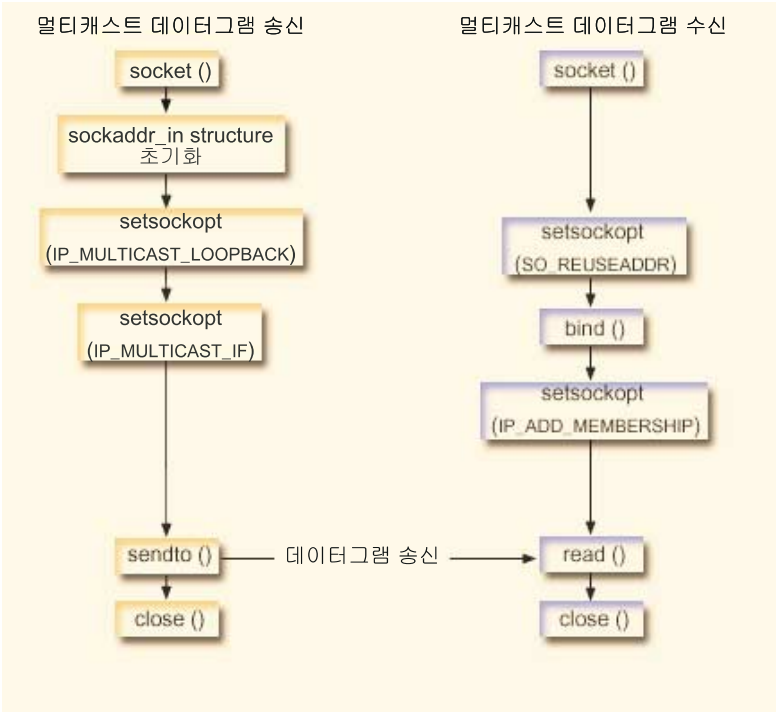
예: 멀티캐스팅 사용

IP 멀티캐스팅은 어플리케이션이 네트워크의 호스트 그룹이 수신할 수 있는 단일 IP 데이터그램을 송신할 수 있는 기능을 제공합니다. 그룹에 있는 호스트는 단일 서브네트에 있을 수도 있고 멀티캐스트 가능 라우터를 연결하는 서로 다른 서브네트에 있을 수도 있습니다. 호스트는 언제라도 그룹에 가입하거나 그룹에서 탈퇴할 수 있습니다. 호스트 그룹 내에서는 멤버 번호나 위치에 제한이 없습니다. 범위 224.0.0.1에서 239.255.255.255까지의 클래스 D 인터넷 주소는 호스트 그룹을 식별합니다.

어플리케이션 프로그램은 **socket()** API 및 무접속 SOCK_DGRAM 유형 소켓을 사용하여 멀티캐스트 데이터그램을 송신하거나 수신할 수 있습니다. 멀티캐스팅은 일 대 다(one-to-many) 전송 방법입니다. 멀티캐스팅에 대해 연결 지향 소켓 유형 SOCK_STREAM을 사용할 수 없습니다. 소켓 유형이 SOCK_DGRAM으로 작성되면 어플리케이션은 **setsockopt()** 함수를 사용하여 해당 소켓과 관련된 멀티캐스트 특성을 제어할 수 있습니다. setsockopt() 함수는 다음 IPPROTO_IP 레벨 플래그를 허용합니다.

- IP_ADD_MEMBERSHIP: 지정된 멀티캐스트 그룹에 가입
- IP_DROP_MEMBERSHIP: 지정된 멀티캐스트 그룹에서 탈퇴
- IP_MULTICAST_IF: 송신 멀티캐스트 데이터그램이 전송될 인터페이스 설정
- IP_MULTICAST_TTL: 송신 멀티캐스트 데이터그램에 대해 IP 헤더에 TTL(생존 시간) 설정
- IP_MULTICAST_LOOP: 멀티캐스트 그룹 멤버로 있는 한 송신 멀티캐스트 데이터그램 사본을 송신 호스트로 전달해야 하는지를 지정

주: OS/400 소켓은 AF_INET 주소 패밀리에 IP 멀티캐스팅을 지원합니다.



이벤트의 소켓 흐름: 멀티캐스트 데이터그램 송신

다음 소켓 호출 순서는 그래픽에 대한 설명을 제공합니다. 멀티캐스트 데이터그램을 송/수신하는 두 어플리케이션간의 관계도 설명합니다. 각각의 흐름 세트에는 특정 API에서의 사용법 노트로의 링크가 포함되어 있습니다. 특정 API 사용에 대한 추가 정보가 필요하면 이런 링크를 사용할 수 있습니다. 멀티캐스트 데이터그램 송신에서는 다음 기능 호출 순서를 사용합니다.

1. **socket()** 함수는 종료점을 표시하는 소켓 설명자를 리턴합니다. 이 명령문은 TCP 전송(SOCK_DGRAM)을 사용하는 INET(인터넷 프로토콜) 주소 패밀리가 이 소켓에 사용된다는 것도 식별합니다. 이 소켓은 다른 어플리케이션으로 데이터그램을 송신합니다.

2. sockaddr_in 구조는 목적지 IP 주소 및 포트 번호를 지정합니다. 이 예에서 주소는 225.1.1.1이고 포트 번호는 5555입니다.
3. **setsockopt()** 함수는 IP_MULTICAST_LOOP 소켓 옵션을 설정하여 송신 시스템이 전송하는 멀티캐스트 데이터그램 사본을 수신하지 않도록 합니다.
4. **setsockopt()** 함수는 멀티캐스트 데이터그램이 송신되는 로컬 인터페이스를 정의하는 IP_MULTICAST_IF 소켓 옵션을 사용합니다.
5. **sendto()** 함수는 멀티캐스트 데이터그램을 지정된 그룹 IP 주소로 송신합니다.
6. **close()** 함수는 열린 소켓 설명자를 닫습니다.

이벤트의 소켓 흐름: 멀티캐스트 데이터그램 수신

멀티캐스트 데이터그램 수신에서는 다음 기능 호출 순서를 사용합니다.

1. **socket()** 함수는 종료점을 표시하는 소켓 설명자를 리턴합니다. 이 명령문은 TCP 전송(SOCK_DGRAM)을 사용하는 INET(인터넷 프로토콜) 주소 패밀리가 이 소켓에 사용된다는 것도 식별합니다. 이 소켓은 다른 어플리케이션으로 데이터그램을 송신합니다.
2. **setsockopt()** 함수는 SO_REUSEADDR 옵션을 설정하여 복수 어플리케이션이 동일한 로컬 포트 번호로 지정되는 데이터그램을 수신할 수 있도록 합니다.
3. **bind()** 함수는 로컬 포트 번호를 지정합니다. 이 예에서 멀티캐스트 그룹으로 주소 지정되는 데이터그램을 수신하기 위해 IP 주소는 INADDR_ANY로 지정됩니다.
4. **setsockopt()** 함수는 데이터그램을 수신하는 멀티캐스트 그룹을 결합하는 IP_ADD_MEMBERSHIP 소켓 옵션을 사용합니다. 그룹에 가입할 때 클래스 D 그룹 주소는 로컬 인터페이스의 IP 주소와 함께 지정되어야 합니다. 시스템은 멀티캐스트 데이터그램을 수신하는 각 로컬 인터페이스에 대해 IP_ADD_MEMBERSHIP 소켓 옵션을 호출해야 합니다. 이 예에서 멀티캐스트 그룹(225.1.1.1)은 로컬 9.5.1.1 인터페이스에서 결합됩니다.

   주: IP_ADD_MEMBERSHIP 옵션은 멀티캐스트 데이터그램이 수신될 각 로컬 인터페이스에 대해 호출되어야 합니다.
5. **read()** 함수는 송신되고 있는 멀티캐스트 데이터그램을 읽습니다.
6. **close()** 함수는 열린 소켓 설명자를 닫습니다.

예: 멀티캐스트 데이터그램 송신

다음 예에서는 소켓이 아래에 나열된 단계를 수행하고 멀티캐스트 데이터그램을 송신할 수 있습니다. 이 프로그램에 대해 이벤트의 소켓 흐름에 대한 설명을 검토하려면, 예: 멀티캐스팅 사용을 참조하십시오.

코드 예를 사용하는 것에 대한 정보는 코드 면책사항을 참조하십시오.

```
#include <sys/types.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <netinet/in.h>
#include <stdio.h>
#include <stdlib.h>
```

```
struct in_addr        localInterface;
struct sockaddr_in    groupSock;
int                   sd;
int                   datalen;
char                  databuf[1024];

int main (int argc, char *argv[])
{

  /* ----------------------------------------------------------------*/
  /*                                                                 */
  /* Send Multicast Datagram code example.                           */
  /*                                                                 */
  /* ----------------------------------------------------------------*/

  /*
   * Create a datagram socket on which to send.
   */
  sd = socket(AF_INET, SOCK_DGRAM, 0);
  if (sd < 0) {
    perror("opening datagram socket");
      exit(1);
  }

  /*
   * Initialize the group sockaddr structure with a
   * group address of 225.1.1.1 and port 5555.
   */
  memset((char *) &groupSock, 0, sizeof(groupSock));
  groupSock.sin_family = AF_INET;
  groupSock.sin_addr.s_addr = inet_addr("225.1.1.1");
  groupSock.sin_port = htons(5555);

  /*
   * Disable loopback so you do not receive your own datagrams.
   */
  {
    char loopch=0;

    if (setsockopt(sd, IPPROTO_IP, IP_MULTICAST_LOOP,
                   (char *)&loopch, sizeof(loopch)) < 0) {
      perror("setting IP_MULTICAST_LOOP:");
            close(sd);
      exit(1);
    }
  }

  /*
   * Set local interface for outbound multicast datagrams.
   * The IP address specified must be associated with a local,
   * multicast-capable interface.
   */
  localInterface.s_addr = inet_addr("9.5.1.1");
  if (setsockopt(sd, IPPROTO_IP, IP_MULTICAST_IF,
                 (char *)&localInterface,
                 sizeof(localInterface)) < 0) {
    perror("setting local interface");
```

```
      exit(1);
  }


  /*
   * Send a message to the multicast group specified by the
   * groupSock sockaddr structure.
   */
  datalen = 10;
  if (sendto(sd, databuf, datalen, 0,
             (struct sockaddr*)&groupSock,
             sizeof(groupSock)) < 0)
  {
    perror("sending datagram message");
  }
}
```

예: 멀티캐스트 데이터그램 수신

다음 예는 소켓이 아래 나열된 단계를 수행하여 멀티캐스트 데이터그램을 수신할 수 있도록 합니다. 이 프로그램에 대해 이벤트의 소켓 흐름에 대한 설명을 검토하려면, 예: 멀티캐스팅 사용을 참조하십시오.

코드 예를 사용하는 것에 대한 정보는 코드 면책사항을 참조하십시오.

```
#include <sys/types.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <netinet/in.h>
#include <stdio.h>
#include <stdlib.h>



struct sockaddr_in    localSock;
struct ip_mreq        group;
int                   sd;
int                   datalen;
char                  databuf[1024];

int main (int argc, char *argv[])
{

  /* -------------------------------------------------------------*/
  /*                                                              */
  /* Receive Multicast Datagram code example.                     */
  /*                                                              */
  /* -------------------------------------------------------------*/

  /*
   * Create a datagram socket on which to receive.
   */
  sd = socket(AF_INET, SOCK_DGRAM, 0);
  if (sd < 0) {
    perror("opening datagram socket");
      exit(1);
  }
```

```
  /*
   * Enable SO_REUSEADDR to allow multiple instances of this
   * application to receive copies of the multicast datagrams.
   */
  {
    int reuse=1;

    if (setsockopt(sd, SOL_SOCKET, SO_REUSEADDR,
                   (char *)&reuse, sizeof(reuse)) < 0) {
      perror("setting SO_REUSEADDR");
            close(sd);
      exit(1);
    }
  }

  /*
   * Bind to the proper port number with the IP address
   * specified as INADDR_ANY.
   */
  memset((char *) &localSock, 0, sizeof(localSock));
  localSock.sin_family = AF_INET;
  localSock.sin_port = htons(5555);;
  localSock.sin_addr.s_addr  = INADDR_ANY;

  if (bind(sd, (struct sockaddr*)&localSock, sizeof(localSock))) {
    perror("binding datagram socket");
            close(sd);
      exit(1);
  }


  /*
   * Join the multicast group 225.1.1.1 on the local 9.5.1.1
   * interface.  Note that this IP_ADD_MEMBERSHIP option must be
   * called for each local interface over which the multicast
   * datagrams are to be received.
   */
  group.imr_multiaddr.s_addr = inet_addr("225.1.1.1");
  group.imr_interface.s_addr = inet_addr("9.5.1.1");
  if (setsockopt(sd, IPPROTO_IP, IP_ADD_MEMBERSHIP,
                 (char *)&group, sizeof(group)) < 0) {
    perror("adding multicast group");
            close(sd);
      exit(1);
  }

  /*
   * Read from the socket.
   */
  datalen = sizeof(databuf);
  if (read(sd, databuf, datalen) < 0) {
    perror("reading datagram message");
            close(sd);
      exit(1);
  }

}
```

예: DNS 갱신 및 조회

다음 예에서는 정의역명 시스템(DNS)을 조회하고 갱신하는 방법을 보여줍니다.

코드 예를 사용하는 것에 대한 정보는 코드 면책사항을 참조하십시오.

```
/*********************************************************************/
/* This program updates a DNS using a transaction signature (TSIG) to     */
/* sign the update packet.  It then queries the DNS to verify success.    */
/*********************************************************************/

/*********************************************************************/
/* Header files needed for this sample program                       */
/*********************************************************************/
#include <stdio.h>
#include <errno.h>
#include <arpa/inet.h>
#include <resolv.h>
#include <netdb.h>

/*********************************************************************/
/* Declare update records - a zone record, a pre-requisite record, and    */
/* 2 update records                                                  */
/*********************************************************************/
      ns_updrec update_records[] =
{
   {
            {NULL,&update_records[1]},
            {NULL,&update_records[1]},
            ns_s_zn,                  /* a zone record */
            "mydomain.ibm.com.",
            ns_c_in,
            ns_t_soa,
            0,
            NULL,
            0,
            0,
            NULL,
            NULL,
            0
   },
   {
            {&update_records[0],&update_records[2]},
            {&update_records[0],&update_records[2]},
      ns_s_pr,                /* pre-req record */
            "mypc.mydomain.ibm.com.",
            ns_c_in,
            ns_t_a,
            0,
            NULL,
            0,
      ns_r_nxdomain,    /* record must not exist */
            NULL,
            NULL,
            0
   },
   {
      {&update_records[1],&update_records[3]},
```

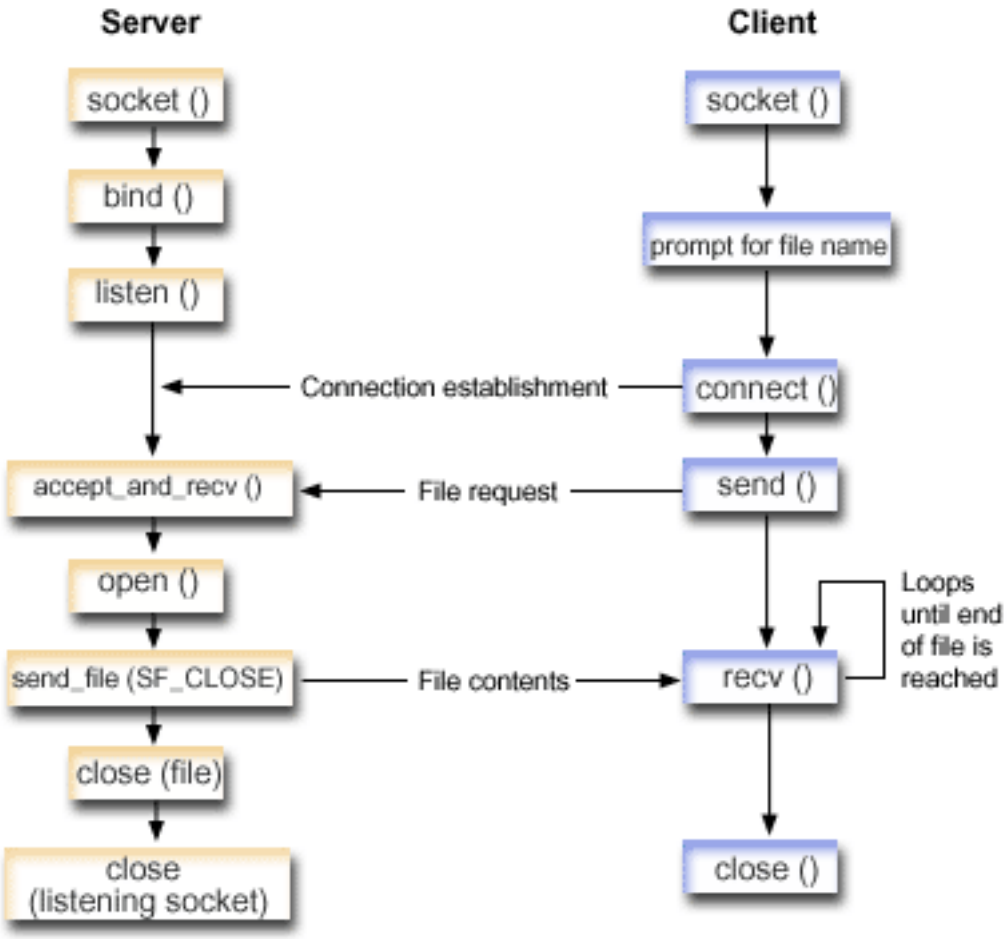
```
      {&update_records[1],&update_records[3]},
      ns_s_ud,                  /* update record */
            "mypc.mydomain.ibm.com.",
            ns_c_in,
      ns_t_a,                   /* IPv4 address  */
            10,
            (unsigned char *)"10.10.10.10",
            11,
      ns_uop_add,                /* to be added  */
            NULL,
            NULL,
            0
   },
   {
      {&update_records[2],NULL},
      {&update_records[2],NULL},
      ns_s_ud,                  /* update record */
            "mypc.mydomain.ibm.com.",
            ns_c_in,
      ns_t_aaaa,                /* IPv6 address  */
            10,
      (unsigned char *)"fedc:ba98:7654:3210:fedc:ba98:7654:3210",
      39,
      ns_uop_add,                /* to be added  */
            NULL,
            NULL,
            0
   }
};

/*************************************************************************/
/* These two structures define a key and secret that must match the one  */
/* configured on the DNS :                                               */
/*   allow-update {                                                      */
/*   key my-long-key.;                                                   */
/*   }                                                                   */
/*                                                                       */
/* This must be the binary equivalent of the base64 secret for           */
/* the key                                                               */
/*************************************************************************/
unsigned char secret[18] =
{
   0x6E,0x86,0xDC,0x7A,0xB9,0xE8,0x86,0x8B,0xAA,
   0x96,0x89,0xE1,0x91,0xEC,0xB3,0xD7,0x6D,0xF8
};

      ns_tsig_key my_key = {
         "my-long-key",           /* This key must exist on the DNS */
         NS_TSIG_ALG_HMAC_MD5,
   secret,
   sizeof(secret)
};

void main()
{
   /*************************************************************************/
  /* Variable and structure definitions. */
   /*************************************************************************/
```

```
struct state res;
int result, update_size;
unsigned char update_buffer[2048];
unsigned char answer_buffer[2048];
int buffer_length = sizeof(update_buffer);

/* Turn off the init flags so that the structure will be initialized */
res.options &= ~ (RES_INIT | RES_XINIT);

result = res_ninit(&res);

/* Put processing here to check the result and handle errors       */

/* Build an update buffer (packet to be sent) from the update records  */
update_size = res_nmkupdate(&res, update_records,
                          update_buffer, buffer_length);

/* Put processing here to check the result and handle errors       */

{
char zone_name[NS_MAXDNAME];
size_t zone_name_size = sizeof zone_name;
struct sockaddr_in s_address;
struct in_addr addresses[1];
int number_addresses = 1;

/* Find the DNS server that is authoritative for the domain            */
/* that we want to update                                              */

result = res_findzonecut(&res, "mypc.mydomain.ibm.com", ns_c_in, 0,
                         zone_name, zone_name_size,
                         addresses, number_addresses);

/* Put processing here to check the result and handle errors       */

/* Check if the DNS server found is one of our regular DNS addresses   */
s_address.sin_addr = addresses[0];
s_address.sin_family = res.nsaddr_list[0].sin_family;
s_address.sin_port = res.nsaddr_list[0].sin_port;
memset(s_address.sin_zero, 0x00, 8);

result = res_nisourserver(&res, &s_address);

/* Put processing here to check the result and handle errors       */

/* Set the DNS address found with res_findzonecut into the res         */
/* structure.  We will send the (TSIG signed) update to that DNS.      */
res.nscount = 1;
res.nsaddr_list[0] = s_address;

/* Send a TSIG signed update to the DNS                                */
   result = res_nsendsigned(&res, update_buffer, update_size,
                            &my_key,
                            answer_buffer, sizeof answer_buffer);

/* Put processing here to check the result and handle errors       */
}
```

```
   /*********************************************************************/
   /* The res_findzonecut(), res_nmkupdate(), and res_nsendsigned()     */
   /* could be replaced with one call to res_nupdate() using            */
   /* update_records[1] to skip the zone record:                        */
   /*                                                                   */
   /* result = res_nupdate(&res, &update_records[1], &my_key);          */
   /*                                                                   */
   /*********************************************************************/
   /*********************************************************************/
   /* Now verify that our update actually worked!                       */
   /* We choose to use TCP and not UDP, so set the appropriate option now    */
   /* that the res variable has been initialized. We also want to ignore  */
   /* the local cache and always send the query to the DNS server.      */
   /*********************************************************************/

   res.options |= RES_USEVC|RES_NOCACHE;

   /* Send a query for mypc.mydomain.ibm.com address records            */
   result = res_nquerydomain(&res,"mypc", "mydomain.ibm.com.", ns_c_in, ns_t_a,
                             update_buffer, buffer_length);

   /* Sample error handling and printing errors                         */
   if (result == -1)
   {
      printf("\nquery domain failed.  result = %d \nerrno: %d: %s \
              \nh_errno: %d: %s",
             result,
             errno, strerror(errno),
             h_errno, hstrerror(h_errno));
   }
   /*********************************************************************/
   /* The output on a failure will be:                                  */
   /*                                                                   */
   /* query domain failed.  result = -1                                 */
   /* errno: 0: There is no error.                                      */
   /* h_errno: 5: Unknown host                                          */
   /*********************************************************************/
  return;
}
```

예: send_file() 및 accept_and_recv() API를 사용하여 파일 자료 전송

다음 예에서는 **send_file()** 및 **accept_and_recv()** API를 사용하여 서버가 클라이언트와 통신할 수 있습니다.



이벤트의 소켓 흐름: 서버 송신 파일 내용

다음 소켓 호출 순서는 그래픽에 대한 설명을 제공합니다. 파일을 송/수신하는 두 어플리케이션간의 관계도 설명합니다. 각각의 흐름 세트에는 특정 API에서의 사용법 노트로의 링크가 포함되어 있습니다. 특정 API 사용에 대한 추가 정보가 필요하면 이런 링크를 사용할 수 있습니다. 예: accept_and_recv() 및 send_file() API를 사용하여 파일 내용 송신에서는 다음 기능 호출 순서를 사용합니다.

1. 서버는 **socket()**, **bind()** 및 **listen()**를 호출하여 청취 소켓을 작성합니다.
2. 서버는 로컬 및 리모트 주소 구조를 초기설정합니다.
3. 서버는 **accept_and_recv()**를 호출하여 수신 연결을 기다리고 첫 번째 자료 버퍼가 이 연결을 거쳐 도착되기를 기다립니다. 이 호출은 수신된 바이트 수와 이 연결에 연관되는 로컬 및 리모트 주소를 리턴합니다. 이 호출은 **accept()**, **getsockname()** 및 **recv()** API의 조합입니다.
4. 서버는 **open()**을 호출하여 이름이 클라이언트 어플리케이션에서 **accept_and_recv()**의 자료로 확보된 파일을 엽니다.

5. **memset()** 함수는 모든 sf_parms 구조의 모든 필드를 초기값 0으로 설정하는 데 사용됩니다. 서버는 파일 설명자 필드를 **open()**이 리턴한 값으로 설정합니다. 그 다음, 서버는 파일 바이트 필드를 -1로 설정합니다. 이것은 서버가 전체 파일을 송신해야함을 나타냅니다. 시스템은 전체 파일을 송신하므로, 파일 오프셋 필드를 할당할 필요가 없습니다.
6. 서버는 **send_file()**을 호출하여 파일의 내용을 전송합니다. **send_file()**은 전체 파일이 송신되거나 인터럽트될 때까지 완료되지 않습니다. **send_file()**은 파일이 완료될 때까지 어플리케이션이 **read()** 및 **send()** 루프로 이동할 필요가 없으므로 더 효율적입니다.
7. 서버는 **send_file()** API에서 SF_CLOSE 플래그를 지정합니다. SF_CLOSE 플래그는 파일과추적 버퍼(지정한 경우)의 마지막 바이트가 송신되고 나면 자동으로 소켓 연결을 닫아야 한다고 **send_file()** API에 알립니다. SF_CLOSE 플래그를 지정할 경우 어플리케이션에서는 **close()**를 호출할 필요가 없습니다.

이벤트의 소켓 흐름: 클라이언트의 파일 요구

예: 클라이언트의 파일 요구에서는 다음 기능 호출 순서를 사용합니다.

1. 이 클라이언트 프로그램은 0 - 2개의 매개변수를 사용합니다.

   첫 번째 매개변수(지정한 경우)는 서버 어플리케이션이 있는 점 십진 IP 주소나 호스트명입니다.

   두 번째 매개변수(지정한 경우)는 클라이언트가 서버에서 확보하려고 하는 파일의 이름입니다. 서버 어플리케이션은 지정된 파일의 내용을 클라이언트로 보냅니다. 사용자가 어떠한 매개변수도 지정하지 않으면 클라이언트는 서버의 IP 주소에 INADDR_ANY를 사용합니다. 사용자가 두 번째 매개변수를 지정하지 않을 경우 프로그램은 파일명을 입력하라는 프롬트를 표시합니다.
2. 클라이언트는 **socket()**을 호출하여 소켓 설명자를 작성합니다.
3. 클라이언트는 **connect()**를 호출하여 서버에 대한 연결을 설정합니다. 1단계에서는 서버의 IP 주소를 확보하였습니다.
4. 클라이언트는 **send()**를 호출하여 확보하려고 하는 파일명을 서버에 알립니다. 1단계에서는 파일의 이름을 확보하였습니다.
5. 클라이언트는 파일 끝에 도달할 때까지 **recv()**를 호출하는 ″do″ 루프로 이동합니다. **recv()**에서 리턴 코드 0은 서버가 연결을 닫았다는 것을 의미합니다.
6. 클라이언트는 **close()**를 호출하여 소켓을 닫습니다.

예: 파일 내용을 송신할 accept_and_recv() 및 send_file() API 사용

다음 예에서는 **send_file()** 및 **accept_and_recv()** API를 사용하여 서버가 클라이언트와 통신하기 위해 다음에 나열된 단계를 수행할 수 있습니다.

코드 예를 사용하는 것에 대한 정보는 코드 면책사항을 참조하십시오.

```
/************************************************/
/* Server example send file data to client      */
/************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <fcntl.h>
```

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

#define SERVER_PORT 12345

main (int argc, char *argv[])
{
   int    i, num, rc, flag = 1;
   int    fd, listen_sd, accept_sd = -1;

   size_t local_addr_length;
   size_t remote_addr_length;
   size_t total_sent;

   struct sockaddr_in   addr;
   struct sockaddr_in   local_addr;
   struct sockaddr_in   remote_addr;
   struct sf_parms      parms;

   char   buffer[255];

   /*************************************************/
   /* If an argument is specified, use it to       */
   /* control the number of incoming connections    */
   /*************************************************/
   if (argc >= 2)
      num = atoi(argv[1]);
   else
      num = 1;

   /*************************************************/
   /* Create an AF_INET stream socket to receive    */
   /* incoming connections on                       */
   /*************************************************/
   listen_sd = socket(AF_INET, SOCK_STREAM, 0);
   if (listen_sd < 0)
   {
    perror("socket() failed");
          exit(-1);
   }

   /*************************************************/
   /* Set the SO_REUSEADDR bit so that you do not   */
   /* have to wait 2 minutes before restarting      */
   /* the server                                    */
   /*************************************************/
   rc = setsockopt(listen_sd,
                   SOL_SOCKET,
                   SO_REUSEADDR,
                   (char *)&flag,
                   sizeof(flag));
      if (rc < 0)
   {
      perror("setsockop() failed");
            close(listen_sd);
          exit(-1);
   }
```

```
|     /*********************************************/
|     /* Bind the socket                           */
|     /*********************************************/
|     memset(&addr, 0, sizeof(addr));
|     addr.sin_family = AF_INET;
|        addr.sin_addr.s_addr = htonl(INADDR_ANY);
|     addr.sin_port   = htons(SERVER_PORT);
|     rc = bind(listen_sd,
|               (struct sockaddr *)&addr, sizeof(addr));
|        if (rc < 0)
|     {
|      perror("bind() failed");
|              close(listen_sd);
|            exit(-1);
|     }
|
|     /*********************************************/
|     /* Set the listen backlog                    */
|     /*********************************************/
|     rc = listen(listen_sd, 5);
|        if (rc < 0)
|     {
|      perror("listen() failed");
|              close(listen_sd);
|            exit(-1);
|     }
|
|     /*********************************************/
|     /* Initialize the local and remote addr lengths  */
|     /*********************************************/
|     local_addr_length  = sizeof(local_addr);
|     remote_addr_length = sizeof(remote_addr);
|
|     /*********************************************/
|     /* Inform the user that the server is ready      */
|     /*********************************************/
|     printf("The server is ready\n");
|
|     /*********************************************/
|     /* Go through the loop once for each connection  */
|     /*********************************************/
|     for (i=0; i < num; i++)
|     {
|        /*********************************************/
|     /* Wait for an incoming connection               */
|        /*********************************************/
|        printf("Iteration: %d\n", i+1);
|        printf("  waiting on accept_and_recv()\n");
|
|        rc = accept_and_recv(listen_sd,
|                             &accept_sd,
|                             (struct sockaddr *)&remote_addr,
|                             &remote_addr_length,
|                             (struct sockaddr *)&local_addr,
|                             &local_addr_length,
|                             &buffer,
|                             sizeof(buffer));
|        if (rc < 0)
```

```
|        {
|           perror("accept_and_recv() failed");
|              close(listen_sd);
|        close(accept_sd);
|            exit(-1);
|        }
|        printf("  Request for file: %s\n", buffer);
|
|        /*********************************************/
|        /* Open the file to retrieve                 */
|        /*********************************************/
|        fd = open(buffer, O_RDONLY);
|        if (fd < 0)
|        {
|           perror("open() failed");
|              close(listen_sd);
|        close(accept_sd);
|            exit(-1);
|        }
|
|        /*********************************************/
|        /* Initialize the sf_parms structure          */
|        /*********************************************/
|        memset(&parms, 0, sizeof(parms));
|        parms.file_descriptor = fd;
|        parms.file_bytes      = -1;
|
|        /*********************************************/
|        /* Initialize the counter of the total number */
|        /* of bytes sent                              */
|        /*********************************************/
|        total_sent = 0;
|
|        /*********************************************/
|        /* Loop until the entire file has been sent   */
|        /*********************************************/
|     do
|        {
|           rc = send_file(&accept_sd, &parms, SF_CLOSE);
|        if (rc < 0)
|           {
|              perror("send_file() failed");
|        close(fd);
|              close(listen_sd);
|        close(accept_sd);
|            exit(-1);
|           }
|           total_sent += parms.bytes_sent;
|
|        } while (rc == 1);
|
|        printf("  Total number of bytes sent: %d\n", total_sent);
|
|        /*********************************************/
|        /* Close the file that is sent out           */
|        /*********************************************/
|        close(fd);
|     }
```

```
    /*************************************************/
    /* Close the listen socket                       */
    /*************************************************/
             close(listen_sd);

    /*************************************************/
    /* Close the accept socket                       */
    /*************************************************/
    if (accept_sd != -1)
       close(accept_sd);
}
```

예: 파일에 대한 클라이언트 요구

다음 예에서 클라이언트는 서버에서 파일을 요구하고 해당 파일의 내용을 다시 송신하기를 기다릴 수 있습니다.

코드 예를 사용하는 것에 대한 정보는 코드 면책사항을 참조하십시오.

```
/*************************************************/
/* Client example requests file data from server */
/*************************************************/
#include <ctype.h>
#include <stdio.h>
#include <stdlib.h>
#include <netdb.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

#define SERVER_PORT 12345

main (int argc, char *argv[])
{
   int    rc, sockfd;

   char   filename[256];
   char   buffer[32 * 1024];

   struct sockaddr_in   addr;
   struct hostent      *host_ent;

   /*************************************************/
   /* Initialize the socket address structure       */
   /*************************************************/
   memset(&addr, 0, sizeof(addr));
   addr.sin_family = AF_INET;
   addr.sin_port   = htons(SERVER_PORT);

   /*************************************************/
   /* Determine the host name and IP address of the */
   /* machine the server is running on              */
   /*************************************************/
   if (argc < 2)
   {
      addr.sin_addr.s_addr = htonl(INADDR_ANY);
```

```
|     }
|     else if (isdigit(*argv[1]))
|     {
|        addr.sin_addr.s_addr = inet_addr(argv[1]);
|     }
|     else
|     {
|        host_ent = gethostbyname(argv[1]);
|        if (host_ent == NULL)
|        {
|           printf("Host not found!\n");
|            exit(-1);
|        }
|        memcpy((char *)&addr.sin_addr.s_addr,
|               host_ent->h_addr_list[0],
|               host_ent->h_length);
|     }
|
|     /************************************************/
|     /* Check to see if the user specified a filename */
|     /* on the command line                          */
|     /************************************************/
|     if (argc == 3)
|     {
|        strcpy(filename, argv[2]);
|     }
|     else
|     {
|        printf("Enter the name of the file:\n");
|        gets(filename);
|     }
|
|     /************************************************/
|     /* Create an AF_INET stream socket              */
|     /************************************************/
|     sockfd = socket(AF_INET, SOCK_STREAM, 0);
|     if (sockfd < 0)
|     {
|      perror("socket() failed");
|            exit(-1);
|     }
|     printf("Socket completed.\n");
|
|     /************************************************/
|     /* Connect to the server                        */
|     /************************************************/
|     rc = connect(sockfd,
|                  (struct sockaddr *)&addr,
|                  sizeof(struct sockaddr_in));
|        if (rc < 0)
|     {
|      perror("connect() failed");
|           close(sockfd);
|            exit(-1);
|     }
|     printf("Connect completed.\n");
|
|     /************************************************/
```

```
    /* Send the request over to the server           */
    /*************************************************/
    rc = send(sockfd, filename, strlen(filename) + 1, 0);
       if (rc < 0)
    {
       perror("send() failed");
          close(sockfd);
           exit(-1);
    }
    printf("Request for %s sent\n", filename);

    /*************************************************/
    /* Receive the file from the server              */
    /*************************************************/
    do
    {
       rc = recv(sockfd, buffer, sizeof(buffer), 0);
       if (rc < 0)
       {
       perror("recv() failed");
          close(sockfd);
           exit(-1);
       }
         else if (rc == 0)
       {
          printf("End of file\n");
          break;
       }
       printf("%d bytes received\n", rc);

    } while (rc > 0);

    /*************************************************/
    /* Close the socket                              */
    /*************************************************/
          close(sockfd);
 }
```

제 11 장 Xsockets 툴

Xsocket 툴은 iSeries와 함께 제공되는 다수의 툴 중 하나입니다. 모든 툴은 QUSRTOOL 라이브러리에 저장됩니다. Xsocket을 사용하여 프로그래머는 소켓 API에 대해 대화식으로 작업할 수 있습니다. Xsocket 툴을 사용하면 다음 타스크를 수행할 수 있습니다.

- 소켓 API에 대해 학습
- 특정 시나리오를 대화식으로 재작성하여 디버그 지원

주: Xsockets 툴은 ″as-is″ 형식으로 제공됩니다.

Xsocket 사용에 대한 전제조건

- ILE C/400 언어가 설치됩니다.
- 사용권 프로그램 5722-SS1 ‘Openness Include’가 설치됩니다.
- 사용권 프로그램 5722-DG1 ‘IBM HTTP Server’가 설치됩니다.

  주: 웹 브라우저에서 Xsocket을 사용할 경우에 필요합니다.
- 사용권 프로그램 5722-JV1 ‘Developer Kit for Java7csq;가 설치됩니다.

  주: 웹 브라우저에서 Xsocket을 사용할 경우에 필요합니다.

Xsocket 툴을 설치하여 사용하려면 다음 주제를 사용하십시오.

Xsocket 구성
이 주제에서는 소켓 프로그램을 설계하여 컴파일하는 데 도움을 주는 Xsocket 툴을 작성하는 방법에 대해 설명합니다.

Xsocket 사용
이 주제에서는 Xsockets 툴을 사용하는 방법에 대해 설명합니다.

Xsocket 사용자 정의
이 주제에서는 Xsocket 툴을 사용자가 정의하는 방법에 대해 설명합니다.

Xsocket 구성

작성할 수 있는 툴 버전이 두 개 있습니다. 첫 번째 버전은 고유 iSeries 클라이언트입니다. 고유 버전은 전적으로 첫 번째 지침 세트에 의해 작성됩니다. 두 번째 버전은 웹 브라우저를 클라이언트로 사용합니다. 웹 브라우저 클라이언트를 사용하려면 먼저 고유 버전에 대한 고유 설정 지침을 완료해야 합니다.

Xsocket 툴을 작성하려면 다음 단계를 완료하십시오.

1. 툴 패키지를 풀려면 다음을 입력하십시오.

```
CALL QUSRTOOL/UNPACKAGE ('*ALL      ' 1)
```

명령행에 다음을 입력하십시오.

주: 왼쪽 '과 오른쪽 ' 사이에는 10자가 있어야 합니다.

2. 라이브러리 리스트에 QUSRTOOL을 추가하려면 다음을 입력하십시오.

```
ADDLIBLE QUSRTOOL
```

명령행에 다음을 입력하십시오.

3. 다음을 입력하여 Xsocket 프로그램 파일을 작성할 라이브러리를 작성하십시오.

```
CRTLIB <library-name>
```

명령행에 다음을 입력하십시오. <library-name>은 Xsocket 툴 오브젝트가 작성될 라이브러리입니다. 예를 들어,

```
CRTLIB MYXSOCKET
```

은 유효한 라이브러리명입니다.

주: QUSRTOOL 라이브러리에 Xsocket 툴 오브젝트를 추가하지 마십시오. 추가하면 해당 디렉토리 내에서 다른 툴 사용을 방해할 수 있습니다.

4. 라이브러리 리스트에 이 라이브러리를 추가하려면 다음을 입력하십시오.

```
ADDLIBLE <library-name>
```

명령행에 다음을 입력하십시오. <library-name>은 3단계에서 작성한 라이브러리입니다. 예를 들어, MYXSOCKET을 라이브러리명으로 사용한 경우 다음을 입력하십시오.

```
ADDLIBLE MYXSOCKET
```

5. 다음을 입력하여 Xsocket 툴을 자동으로 설치할 설치 프로그램 TSOCRT를 작성하십시오.

```
CRTCLPGM <library-name>/TSOCRT QUSRTOOL/QATTCL
```

명령행에 다음을 입력하십시오. <library-name>은 3단계에서 작성한 라이브러리입니다. 예를 들어, MYXSOCKET을 라이브러리명으로 사용한 경우 다음을 입력하십시오.

```
CRTCLPGM MYXSOCKET/TSOCRT QUSRTOOL/QATTCL
```

6. 설치 프로그램을 호출하려면 다음을 입력하십시오.

```
CALL TSOCRT library-name
```

명령행에 다음을 입력하십시오. library-name 자리에 3단계에서 작성한 라이브러리를 사용하십시오. 예를 들어, MYXSOCKET 라이브러리에서 툴을 작성하려면 다음을 입력하십시오.

```
CALL TSOCRT MYXSOCKET
```

주: 완료되려면 몇 분이 소요됩니다.

소켓 툴 TSOCRT를 호출할 때 작업 제어(*JOBCTL) 특수 권한이 없으면 사용자가 실행하고 있는 작업에 대한 설명자를 패스하려할 때 **givedescriptor()** 소켓이 오류를 리턴합니다.

TSOCRT는 하나의 CL 프로그램, 하나의 ILE C/400 프로그램(두 개의 모듈이 작성됨), 두 개의 ILE C/400 서비스 프로그램(두 개의 모듈이 작성됨) 및 세 개의 화면 파일을 작성합니다. 툴을 사용하려고 할 때마다 라이브러리 리스트에 라이브러리를 추가해야 합니다. 툴로 작성된 모든 오브젝트에는 접두부가 TSO인 이름이 있습니다.

다음 단계:

고유 Xsocket 사용
Xsocket을 고유하게 사용하려는 경우 이 단계로 찾아갈 수 있습니다. 이 주제에서는 고유 Xsocket 툴 사용에 대한 기본적인 내용을 다룹니다.

주: 고유 버전은 GSKit 보안 소켓 API를 지원하지 않습니다. 이 API를 사용하는 소켓 프로그램을 작성하려는 경우 브라우저 기반의 툴 버전을 사용해야 합니다.

웹 브라우저를 사용하기 위한 Xsocket 구성

주: 이 단계는 선택적입니다.

고유 Xsocket 설정으로 작성되는 사항

다음은 설치 프로그램에 의해 작성된 오브젝트를 나열하는 표입니다. 작성된 오브젝트는 모두 지정된 라이브러리에 상주합니다.

표 20. Xsocket을 설치하는 동안 작성된 오브젝트

오브젝트명	멤버명	소스 파일명	오브젝트 유형	확장자	설명
TSOJNI	TSOJNI	QATTSYSC	*MODULE	C	JSP와 TSOSTSOC 간 인터페이싱에 사용되는 모듈
TSODLT	TSODLT	QATTCL	*PGM	CLP	툴 오브젝트 및/또는 소스 파일 멤버를 삭제하기 위한 CL 프로그램.
TSOXSOCK	N/A	N/A	*PGM	C	SOCKETS 대화식 툴에 사용되는 기본 프로그램.
TSOXGJOB	N/A	N/A	*SRVPGM	C	SOCKETS 대화식 툴을 지원하는 데 사용되는 서비스 프로그램.
TSOJNI	N/A	N/A	*SRVPGM	C	SOCKETS 대화식 툴을 지원하여 JSP와 TSOSTSOC간 인터페이싱에 사용되는 서비스 프로그램.

표 20. Xsocket을 설치하는 동안 작성된 오브젝트 (계속)

TSOXSOCK	TSOXSOCK	QATTSYSC	*MODULE	C	TSOXSOCK 프로그램 작성에 사용되는 모듈. 소스 파일은 main() 루틴을 포함합니다.
TSOSTSOC	TSOSTSOC	QATTSYSC	*MODULE	C	TSOXSOCK 프로그램 작성에 사용되는 모듈. 소스 파일은 실제로 소켓 함수를 호출하는 루틴을 포함합니다.
TSOXGJOB	TSOXGJOB	QATTSYSC	*MODULE	C	TSOXGJOB 서비스 프로그램 작성에 사용되는 모듈. 소스 파일은 내부 작업을 식별하는 루틴을 포함합니다. 이 내부 작업 식별자는 작업명, 사용자 ID, 작업 번호로 구성됩니다.
TSODSP	TDSPDSP	QATTDDS	*FILE	DSPF	소켓 함수를 포함하는 기본 창에 대해 Xsocket 툴이 사용하는 화면 파일.
TSOFUN	TDSOFUN	QATTDDS	*FILE	DSPF	여러 소켓 함수를 지원하여 XSocket 툴이 사용하는 화면 파일.
TSOMNU	TDSOMNU	QATTDDS	*FILE	DSPF	메뉴 바를 지원하는 Xsocket 툴이 사용하는 화면 파일.
QATTIFS2	N/A	N/A	*FILE	PF-DTA	Tomcat Server에 사용되는 JAR 파일을 포함합니다.

웹 브라우저를 사용하기 위한 Xsocket 구성

다음 지침 세트를 사용하면 웹 브라우저를 통해 Xsocket 툴에 액세스할 수 있습니다. 동일한 시스템에서 이 지침을 여러 번 구현하여 서로 다른 서버 인스턴스를 작성할 수 있습니다. 이렇게 하면 서로 다른 청취 포트에서 동시에 복수 버전을 실행할 수 있습니다. 웹 브라우저를 사용하기 위해 Xsocket을 구성하려면 다음 타스크를 완료해야 합니다.

1. HTTP Server(Apache로 구동) 구성
2. Tomcat 구성
3. 구성 파일 갱신
4. 브라우저에서 Xsocket 툴 테스트

HTTP Server(Apache로 구동) 구성

Xsocket 툴에 대한 작업을 하기 위해 웹 브라우저를 구성하기 전에 먼저 고유 Xsocket 구성을 완료해야 합니다. 다음 단계는 웹 브라우저에서 Xsocket 툴을 사용할 수 있도록 HTTP Server(Apache로 구동)를 구성합니다.

1. HTTP admin 인스턴스가 QHTTPSVR 서브시스템에서 실행되고 있는지 확인하십시오. 실행되고 있지 않은 경우 CL 명령으로 시작할 수 있습니다.

   ```
   STRTCPSVR SERVER(*HTTP) HTTPSVR(*ADMIN)
   ```

2. 웹 브라우저에서 다음을 입력하십시오.

   ```
   http://<system_name>:2001/.
   ```

   여기서 <system_name>은 iSeries의 기계명입니다. 예를 들어, http://myiSeries:2001/입니다.
3. iSeries 타스크 페이지에서 **iSeries용 IBM HTTP Server**를 선택하십시오.
4. 최상위 메뉴에서 **설정** 탭을 선택하십시오.
5. **새 HTTP Server 작성**.
6. **HTTP Server(Apache로 구동)**을 선택하고 **다음**을 클릭하십시오.
7. 서버 인스턴스에 이름을 입력하십시오. 예를 들면 이 인스턴스가 브라우저에서 Xsocket 툴에 서비스를 제공할 것이므로 xsocket이라는 이름을 사용할 수 있습니다. **다음**을 클릭하십시오.
8. **아니오**를 선택하십시오. 이렇게 하면 기존 서버를 기반으로 하지 않는 새로운 서버 인스턴스가 작성됩니다. **다음**을 클릭하십시오.
9. 서버 루트 디렉토리에 디폴트를 허용하려면 **다음**을 클릭하십시오.
10. 문서 루트 디렉토리에 디폴트를 허용하려면 **다음**을 클릭하십시오.
11. IP 주소 및 사용하려는 사용 가능 포트를 선택하십시오. 1024보다 큰 숫자의 포트 번호를 사용하십시오. **다음**을 클릭하십시오.

    주: 디폴트 포트 번호 80을 선택하지 마십시오.
12. 이 서버에 대해 작성된 액세스 기록부의 선택 여부를 가리키려면 **예** 또는 **아니오**를 선택하십시오. **다음**을 클릭하십시오.
13. 다음 페이지에 HTTP Server(Apache로 구동) 구성 설정이 표시됩니다. 이 설정이 올바르면 **완료**를 클릭하십시오.
14. **새로 작성된 서버 관리**를 클릭하십시오. 이제 Apache 구성을 완료하였습니다.

다음 단계:

Tomcat 구성

Tomcat 구성

HTTP Server(Apache로 구동) 서버 인스턴스를 구성한 후에는 Tomcat을 구성하여 웹 브라우저에서 Xsocket 툴을 실행해야 합니다.

1. **동적 목차** 머리말 아래에서 **ASF Tomcat 설정 서버 타스크**를 선택하십시오.
2. **이 HTTP Server에 서브릿 작동**을 선택하십시오. 이렇게 하면 작업자 정의 파일이 채워집니다. **다음**을 클릭하십시오.
3. **다음**을 클릭하여 **작업자 정의** 페이지의 디폴트를 허용하십시오.
4. **작업자에 URL 맵핑** 페이지에서 **추가**를 클릭하십시오.
5. **URL(마운트 위치)** 열에 /xsock를 입력하십시오. **계속**을 클릭하십시오.
6. **추가**를 클릭하십시오.
7. **URL(마운트 위치)** 열에 /xsock/*를 입력하십시오. **계속**을 클릭하십시오.
8. **다음**을 클릭하십시오.
9. **처리 중인 어플리케이션 문맥 정의** 페이지에서 **추가**를 클릭하십시오.
10. **URL 경로** 열에 /xsock를 입력하십시오.
11. **어플리케이션 기본 디렉토리** 열에 webapps/xsock를 입력하십시오.
12. **계속**을 클릭하십시오.정보를 더 구성해야 한다는 경고 메세지가 나타납니다.
13. **어플리케이션 구성** 아래의 **구성**을 클릭하십시오.
14. 열리는 새로운 브라우저 창에서 **세션 오브젝트 시간종료** 필드에서 3일을 선택하십시오.

    주: 이것이 권장되는 값입니다. 그러나 **세션 오브젝트 시간종료**에 다른 값을 지정할 수 있습니다.
15. **추가**를 클릭하여 서브릿 정의를 추가하고 다음 단계를 완료하십시오.
    a. **서브릿 클래스명**에 com.ibm.iseries.xsocket.XSocketServlet을 입력하십시오.
    b. **URL 패턴**의 경우 /*를 입력하십시오.
    c. **시작 로드 순서**를 3으로 설정하십시오.
    d. **계속**을 클릭하십시오.
    e. **확인**을 클릭하십시오. 브라우저 창이 닫힙니다.
16. 기본 Tomcat 설정 창에서 **다음**을 클릭하십시오.
17. **완료**를 클릭하십시오.
18. **확인**을 클릭하십시오. 이제 Xsocket 툴에 대한 Tomcat 구성을 완료했습니다.

다음 단계:

구성 파일 갱신

구성 파일 갱신

HTTP Server(Apache로 구동)에서 Xsocket 툴에 서비스를 제공하기 위해 Tomcat 구성을 완료한 후에는 인스턴스에 대한 몇 개의 구성 파일을 수동으로 변경해야 합니다. 세 개의 파일(web.xml 파일, JAR 파일 및 httpd.conf)을 갱신해야 합니다. 이 단계를 완료하려면 다음 정보를 알아야 합니다.

- Xsocket 어플리케이션 파일을 포함하는 라이브러리명. 고유 클라이언트에 초기 Xsocket 구성을 하는 동안 이 파일을 작성했습니다.

- THTTP Server(Apache로 구동) 구성을 하는 동안 작성한 서버명.

1. **web.xml 파일 갱신**
   a. 명령행에서 다음을 입력하십시오.

   ```
   wrklnk '/www/<server_name>/webapps/xsock/WEB-INF/web.xml'
   ```

   여기서 <server_name>은 Apache 구성을 하는 동안 작성한 서버 인스턴스의 이름입니다. 예를 들어, 서버명으로 xsocks를 선택한 경우 다음을 입력하십시오.

   ```
   wrklnk '/www/xsocks/webapps/xsock/WEB-INF/web.xml'
   ```

   b. 2를 눌러 파일을 편집하십시오.
   c. web.xml 파일에서 </servlet-class> 행을 찾으십시오.
   d. 이 행 뒤에 다음 코드를 삽입하십시오.

   ```
   <init-param>
             <param-name>library</param-name>
             <param-value>XXXX</param-value>
   </init-param>
   ```

   XXXX 자리에 Xsocket 구성을 하는 동안 작성한 라이브러리명을 삽입하십시오.
   e. 파일을 저장하고 편집 세션을 나가십시오.

2. **JAR 파일 이동**
   a. 명령행에서 다음 명령을 입력하십시오.

   ```
   CPY OBJ('/QSYS.LIB/XXXX.LIB/QATTIFS2.FILE/TSOXSOCK.MBR')
      TOOBJ('/www/<server_name>/webapps/xsock/WEB-INF/lib/tsoxsock.jar')
      FROMCCSID(*OBJ) TOCCSID(819) OWNER(*NEW)
   ```

   여기서 XXXX는 Xsocket 구성을 하는 동안 작성한 라이브러리명이고 <server_name>은 HTTP Server(Apache로 구동) 구성을 하는 동안 작성한 서버 인스턴스의 이름입니다.

3. **httpd.conf 파일에 권한 검사 추가**(이 단계는 선택적입니다.)
   이것은 Apache가 Xsocket 웹 어플리케이션에 액세스하려고 시도하는 사용자를 강제로 인증하도록 합니다.

   주: UNIX 소켓을 작성하려면 쓰기 액세스도 확보해야 합니다.

   a. 명령행에서 다음을 입력하십시오.

   ```
   wrklnk '/www/<server_name>/conf/httpd.conf'
   ```

   여기서 <server_name>은 Apache 구성을 하는 동안 작성한 서버 인스턴스의 이름입니다. 예를 들어, 서버명으로 xsocks를 선택한 경우 다음을 입력하십시오.

   ```
   wrklnk '/www/xsocks/conf/httpd.conf''
   ```

   b. 2를 눌러 파일을 편집하십시오.
   c. 파일의 맨 끝에 다음 행을 삽입하십시오.

```
<Location /xsock>
   AuthName "X Socket"
   AuthType Basic
   PasswdFile %%SYSTEM%%
   UserId %%CLIENT%%
   Require valid-user
   order allow,deny
   allow from all
</Location>
```

d. 파일을 저장하고 편집 세션을 나가십시오.

다음 단계:

웹 브라우저에서 Xsocket 툴 테스트

웹 브라우저에서 Xsocket 툴 테스트

구성 파일에 대한 수동 갱신을 완료한 후에는 브라우저 내에서 Xsocket 툴을 테스트할 준비가 됩니다.

1. 서버 인스턴스를 시작하려면 명령행에 다음 명령을 입력하십시오.

   ```
   STRTCPSVR SERVER(*HTTP) HTTPSVR(<server_name>)
   ```

   여기서 <server_name>은 Apache 구성을 하는 동안 작성한 서버 인스턴스의 이름입니다.

   주: 몇 분이 소요됩니다.
2. 명령행 인터페이스에서 WRKACTJOB 명령을 발행하여 상태를 검사하십시오. 서버 이름을 가진 모든 작업이 SIGW 상태이면, 다음 단계로 진행할 수 있습니다.
3. 브라우저에서 다음 URL을 입력하십시오.

   ```
   http://<system_name>:<port>/xsock/index
   ```

   여기서 <system_name> 및 <port>는 Apache 구성을 하는 동안 선택한 서버 인스턴스명과 포트 번호입니다.
4. 프롬트되면 서버에 사용자 이름과 암호를 입력하십시오. Xsocket에 대한 웹 클라이언트가 나타나야 합니다.

Xsocket 사용

현재 Xsocket 툴에 대한 작업을 하는 방법은 두 가지입니다. 고유 클라이언트에서 툴에 대한 작업을 하거나 웹 브라우저에서 툴에 대한 작업을 할 수 있습니다. Xsocket 고유 버전에 대한 작업을 하려면 Xsocket 툴을 구성해야 합니다. 브라우저 환경에서 툴에 대한 작업을 하려는 경우 고유 클라이언트에 Xsocket 툴을 구성하는 것 외에 웹 브라우저를 사용하기 위한 Xsocket 구성의 단계도 완료해야 합니다. 두 가지 툴 버전간에 유사한 개념이 많습니다. 두 툴 모두 대화식으로 소켓 호출을 발행할 수 있도록 하며 두 툴 모두 발행된 소켓 호출에 대해 errno를 제공합니다. 그러나 인터페이스는 약간 차이가 있습니다. 다음 지침 세트는 두 환경에서 Xsocket 툴에 대한 작업을 하는 방법을 나타냅니다.

주: GSKit 보안 소켓 API를 사용하는 소켓 프로그램에 대한 작업을 하려면 웹 버전의 툴을 사용해야 합니다.

다음 지침 세트는 이 툴 각각에 대한 작업을 하는 방법을 설명합니다.

- 고유 Xsocket 사용
- 브라우저 기반의 Xsocket 사용

고유 Xsocket 사용

이 단계를 완료하기 전에 고유 Xsocket에 대해 구성 단계를 모두 완료했는지 확인하십시오. 고유 클라이언트에서 Xsocket을 사용하려면 다음 단계를 완료하십시오.

1. 명령행에서 다음 명령을 발행하여 라이브러리 리스트에 Xsocket 툴이 있는 라이브러리를 추가하십시오.

   ```
   ADDLIBLE <library-name>
   ```

   여기서 <library-name>은 고유 Xsocket을 구성하는 동안 작성한 라이브러리의 이름입니다. 예를 들어, 라이브러리명이 MYXSOCKET이면 다음을 입력하십시오.

   ```
   ADDLIBLE MYXSOCKET
   ```

2. 명령행 인터페이스에서 다음을 입력하십시오.

   ```
   CALL TSOXSOCK
   ```

3. 표시되는 Xsocket 창에서 메뉴 바와 선택 필드를 통해 모든 소켓 루틴에 액세스할 수 있습니다. 이 창은 항상 소켓 함수를 선택한 후에 표시됩니다. 이 인터페이스를 사용하여 이미 존재하는 소켓 프로그램을 선택할 수 있습니다. 새로운 소켓에 대한 작업을 하려면 다음을 완료하십시오.
   a. 소켓 함수 리스트에서 **socket**을 선택하고 **Enter**를 누르십시오.
   b. 표시되는 **socket() 프롬트** 창에서 소켓에 적절한 주소 패밀리, 소켓 유형 및 프로토콜을 선택하고 **Enter**를 누르십시오.
   c. **설명자**를 선택하고 **설명자 선택**을 선택하십시오.

      주: 다른 소켓 설명자가 이미 있는 경우 활동 소켓 설명자 리스트가 표시됩니다.

   d. 표시되는 리스트에서 작성한 소켓 설명자를 선택하십시오.

      주: 다른 소켓 설명자가 있는 경우 툴은 최신 소켓 설명자에 소켓 함수를 자동으로 적용합니다.

4. 소켓 함수 리스트에서 작업할 socket 함수를 선택하십시오. 3c단계에서 선택한 소켓 설명자가 해당 소켓 함수에 사용됩니다. 일단 소켓 함수를 선택하면 소켓 함수에서 특정 정보를 제공할 수 있는 일련의 창이 표시됩니다. 예를 들어, **connect()**를 선택하면 결과 창에서 주소 길이, 주소 패밀리 및 주소 자료를 제공해야 합니다. 그러면 선택한 소켓 함수는 제공된 이 정보와 함께 호출됩니다. 소켓 함수에서 발생한 모든 오류는 errno로 사용자에게 다시 표시됩니다.

주:

1. Xsocket 툴은 DDS에 대한 그래픽 보기 지원을 사용합니다. 따라서 자료가 입력되는 방법과 사용자가 보게 되는 창/패널에서 선택하는 방법은 그래픽 표시장치를 사용하고 있는지 비그래픽 표시장치를 사용하고 있는지에 따라 다릅니다. 예를 들어, 그래픽 표시장치에서는 소켓 함수에 대한 선택 필드를 선택란으로 보게 됩니다. 아니면 단일 필드를 볼 수 있습니다.
2. 툴에서 구현되지 않은 소켓에서 사용할 수 있는 **ioctl()** 요구가 있다는 것을 알아 두십시오.

브라우저에서 Xsocket 사용

웹 브라우저에서 Xsocket에 대한 작업을 시작하기 전에 Xsocket 고유 구성 및 필요한 웹 브라우저 구성을 모두 완료했는지 확인하십시오. 웹 브라우저에서 Xsocket에 대한 작업을 하려면 다음 단계를 완료하십시오.

1. 웹 브라우저에서 다음을 입력하십시오.

   ```
   http://server-name:2001/
   ```

   여기서 server-name은 서버 인스턴스를 포함하는 iSeries의 이름입니다.
2. **관리**를 선택하십시오.
3. 왼쪽 검색에서 **HTTP Server 관리**를 선택하십시오.
4. 인스턴스명을 선택하고 **시작**을 클릭하십시오. 명령행에서 다음을 입력하여 서버 인스턴스를 시작할 수도 있습니다.

   ```
   STRTCPSVR SERVER(*HTTP) HTTPSVR(<instance_name>)
   ```

   여기서 <instance_name>은 Apache 구성에서 작성한 HTTP Server의 이름입니다. 예를 들어, 서버 인스턴스명으로 xsocks를 사용할 수 있습니다.
5. Xsocket 웹 어플리케이션에 액세스하려면 브라우저에서 다음 URL을 입력하십시오.

   ```
   http://<system_name>:<port>/xsock/index
   ```

   여기서 <system_name>은 iSeries의 기계명이고 <port>는 HTTP 인스턴스를 작성할 때 지정한 포트입니다. 예를 들어, 시스템명이 myiSeries이고 HTTP 서버 인스턴스가 1025 포트에서 청취하는 경우 다음을 입력하십시오.

   ```
   http://myiSeries:1025/xsock/index
   ```
6. 웹 브라우저에 Xsocket 툴이 로드되면 기존의 소켓 설명자에 대한 작업을 하거나 새로운 소켓 설명자를 작성할 수 있습니다. 두 툴 버전간에 유사한 개념이 많습니다. 두 툴 모두 대화식으로 소켓 호출을 발행할 수 있도록 하며 두 툴 모두 발행된 소켓 호출에 대해 errno를 제공합니다. 그러나 인터페이스는 약간 차이가 있습니다. 새로운 소켓 설명자를 작성하기 위해 다음을 수행할 수 있습니다.
   a. **Xsocket 메뉴**에서 **소켓**을 선택하십시오.
   b. 표시되는 **Xsocket 조회** 창에서 이 소켓 설명자에 적절한 주소 패밀리, 소켓 유형 및 프로토콜을 선택하십시오. **제출**을 클릭하십시오.
   c. 페이지가 다시 로드되면 새로운 소켓 설명자가 **소켓** 풀다운 메뉴에 표시됩니다.

d. **Xsocket 메뉴**에서 이 소켓 설명자에 적용하려는 기능 호출을 선택하십시오. Xsocket 툴 고유 버전의 경우와 마찬가지로 소켓 설명자를 선택하지 않으면 툴이 최신 소켓 설명자에 기능 호출을 자동으로 적용합니다.

Xsocket 툴이 작성한 오브젝트 삭제

Xsocket 툴이 작성한 오브젝트를 삭제해야 할 수 있습니다. 툴로 작성한 오브젝트를 제거(라이브러리 및 TSODLT 프로그램 제외) 및/또는 Xsocket 툴이 사용하는 소스 멤버를 제거하기 위해 TSODLT라는 프로그램이 설치 프로그램에 의해 작성됩니다. 다음 명령 세트를 사용하면 이런 오브젝트를 삭제할 수 있습니다.

툴이 사용하는 소스 멤버만 삭제하려면 다음 명령을 입력하십시오.

```
CALL TSODLT   (*YES *NONE)
```

툴이 작성하는 오브젝트만 삭제하려면 다음 명령을 입력하십시오.

```
CALL TSODLT   (*NO  library-name)
```

툴이 작성한 오브젝트와 소스 멤버를 모두 삭제하려면 다음 명령을 입력하십시오.

```
CALL TSODLT   (*YES library-name)
```

Xsocket 사용자 정의

inet_addr()와 같은 소켓 네트워크 루틴의 추가 지원을 추가하여 Xsocket 툴을 변경할 수 있습니다. 사용자 자신의 요구에 맞도록 이 툴의 사용자 정의를 선택할 경우 QUSRTOOL 라이브러리에서는 변경하지 않는 것이 바람직합니다. 대신에, 소스 파일을 별도의 라이브러리에 복사한 후 거기에서 변경하십시오. 이는 QUSRTOOL 라이브러리의 원래 파일을 보존하므로, 나중에 필요한 경우 이 파일을 사용할 수 있게 됩니다. 변경 후에는 TSOCRT 프로그램을 사용하여 툴을 다시 컴파일할 수 있습니다(소스 파일이 별도의 라이브러리에 복사된 경우 이 파일을 사용하려면 TSOCRT에서도 변경해야 한다는 점에 유의하십시오). 툴을 작성하기 전에 툴 오브젝트의 이전 버전을 제거하려면 TSODLT 프로그램을 사용하십시오.

제 12 장 서비스 능력 확장 툴

소켓 및 소스 소켓의 사용은 e-business 어플리케이션 및 서버를 조정하기 위해 계속되므로, 현재 서비스 능력 툴은 이러한 요구에 맞아야 합니다. 향상된 서비스 능력 툴을 사용하여 소켓 및 SSL 어플리케이션 내의 오류에 대한 솔루션을 찾기 위해 소켓 프로그램에서 추적을 완료할 수 있습니다. 이러한 툴은 사용자와 지원 센터 담당자가 IP 주소나 포트 정보와 같은 소켓 특징을 선택하여 소켓 문제점이 있는 위치를 판별하는 데 도움을 줍니다.

다음 표에는 이러한 서비스 툴 각각에 대한 개요가 나와 있습니다.

표 21. 소켓 및 보안 소켓에 대한 서비스 능력 확장 툴

서비스 능력 확장 툴	설명
LIC 추적 필터링(TRCINT 및 TRCCNN)	소켓의 선택 추적을 제공합니다. 이제 주소 패밀리, 소켓 유형, 프로토콜, IP 주소 및 포트 정보에 대한 소켓 추적을 제한할 수 있습니다. 소켓 API의 특정 범주로만으로 제한할 수도 있으며 SO_DEBUG 소켓 옵션 세트가 있는 소켓으로만 제한할 수도 있습니다. 이제 V5R2에서는 스레드, 타스크, 사용자 프로파일, 작업명 또는 서비스명별로 LIC 추적을 필터할 수 있습니다.
STRTRC SSNID(*GEN) JOBTRCTYPE(*TRCTYPE) TRCTYPE((*SOCKETS *ERROR))으로 작업 추적	V5R2에서는 STRTRC 명령이 다른 모든 비소켓 관련 추적점과 분리된 출력을 생성하는 매개변수를 추가로 제공합니다. 이 출력에는 소켓 조작을 하는 동안 오류가 발행했을 때 리턴 코드와 errno 정보가 포함되어 있습니다. 세부사항은 Information Center의 STRTRC(추적 시작) 작업 설명을 참조하십시오.
플라이트(flight) 레코더 추적	소켓 LIC 구성요소 추적에는 이제 수행된 각 소켓 조작마다 플라이트 레코더 항목의 덤프가 포함됩니다.
연관 작업 정보	서비스 담당자 및 프로그래머가 연결되거나 청취 소켓과 연관된 모든 작업을 찾을 수 있습니다. 이 정보는 AF_INET 또는 AF_INET6 주소 패밀리를 사용하는 그러한 소켓 어플리케이션에 NETSTAT를 사용하여 열람할 수 있습니다.
SO_DEBUG를 작동하기 위한 NETSTAT 연결 상태(옵션 '3')	SO_DEBUG 소켓 옵션이 소켓 어플리케이션에서 설정되어 있을 때 향상된 하위 레벨 디버그 정보를 제공합니다.
보안 소켓 리턴 코드 및 메세지 처리	두 개의 SSL_ API를 통해 표준화된 보안 소켓 리턴 코드 메세지를 제공합니다. 이러한 API로는 **SSL_Strerror()**과 **SSL_Perror()**가 있습니다. 또한 **gsk_strerror()**는 GSKit API에 유사한 기능을 제공합니다. 분석기 루틴에서 리턴 코드 정보를 제공하는 **hstrerror()** API도 있습니다.
성능 자료 콜렉션(PDC) 추적점	소켓 및 TCP/IP 스택을 통해 어플리케이션에서 자료 흐름에 대한 추적을 제공합니다.

제 13 장 관련된 정보

아래에는 소켓 프로그래밍 추가 정보를 제공하는 IBM 레드북(PDF 형식), 웹 사이트 및 Information Center 주제가 나열되어 있습니다. PDF를 보거나 인쇄할 수 있습니다.

IBM 레드북

- Who Knew You Could Do That with RPG IV? A Sorcerer's Guide to System Access and More

  (약 454 페이지)
- IBM @server iSeries Wired Network Security: OS/400 V5R1 DCM and Cryptographic Enhancements

  (약 506 페이지)

RFC(Request For Comments)

- **IPv6**
  - RFC 2553: ″Basic Socket Interface Extensions for IPv6″
  - RFC 2292: ″Advanced Sockets API for IPv6″
- 정의역명 시스템
  - RFC 1034: ″Domain names - concepts and facilities″
  - RFC 1035: ″Domain names - implementation and specification″
  - RFC 2136: ″Dynamic Updates in the Domain Name System (DNS UPDATE)″
  - RFC 2181: ″Clarifications to the DNS Specification″
  - RFC 2308: ″Negative Caching of DNS Queries (DNS NCACHE)″
  - RFC 2845: ″Secret Key Transaction Authentication for DNS (TSIG)″
- 보안 소켓 층/전송 층 보안
  - RFC 2246: ″The TLS Protocol Version 1.0 ″

기타 웹 자원

- Technical Standard: Networking Services(XNS), Issue 5.2 Draft 2.0. 

제 14 장 코드 면책사항 정보

이 문서에는 프로그래밍 예제가 들어 있습니다.

IBM은 귀하에게 유사한 기능을 귀하의 특정 요구에 맞게 조정하여 생성할 수 있도록 모든 프로그래밍 코드 예제를 사용할 수 있는 비독점적인 저작권 사용권을 부여합니다.

모든 샘플 예제는 IBM에 의해 예시 목적으로만 제공됩니다. 이러한 예제는 모든 조건하에서 철저히 테스트된 것은 아닙니다. 따라서 IBM은 이들 프로그램의 신뢰성, 실용성 또는 기능에 대해 보증할 수 없습니다.

여기에 포함된 모든 프로그램은 상품성 및 특정 목적에의 적합성에 대한 묵시적 보증을 포함하여 어떠한 종류의 보증 없이 "현상태대로" 제공됩니다.

Printed in U.S.A