iSeries

Programowanie z użyciem gniazd

Wersja 5 wydanie 3

eserver

iSeries

Programowanie z użyciem gniazd

Wersja 5 wydanie 3

Uwaga

Przed korzystaniem z tych informacji oraz produktu, którego dotyczą, należy przeczytać informacje znajdujące się w sekcji “Uwagi” na stronie 181.

Wydanie piąte (sierpień 2005)

Niniejsze wydanie dotyczy wersji 5, wydania 3, modyfikacji 0 systemu Operating System/400 (5722–SS1) oraz wszelkich kolejnych wersji i modyfikacji tego produktu, o ile nowe wydania nie wskazują inaczej. Niniejsza wersja nie działa we wszystkich modelach komputerów RISC ani CISC.

© Copyright International Business Machines Corporation 2001, 2005. Wszelkie prawa zastrzeżone.

Spis treści

Programowanie z użyciem gniazd 1
Drukowanie tego dokumentu 2
Wymagania wstępne dla programowania z użyciem gniazd 2
Jak działają gniazda. 3
Charakterystyki gniazd 5
Struktura adresu gniazda 6
Rodzina adresów gniazd 7
Typy gniazd 12
Protokoły gniazd 12
Podstawy projektowania gniazd 13
Tworzenie gniazd zorientowanych na połączenie. . . 13
Tworzenie gniazd bezpołączeniowych 21
Projektowanie aplikacji używających rodzin adresów 26
Pojęcia dotyczące gniazd 45
Asynchroniczne operacje we/wy 46
Gniazda chronione. 48
Obsługa klienta SOCKS 54
Ochrona wątków 56
Nieblokujące operacje we/wy 57
Sygnały 57
Rozsyłanie grupowe IP 58
Przesyłanie danych pliku – funkcje send_file() i accept_and_recv() 59
Dane spoza pasma 59
Multipleksowanie operacji we/wy – funkcja select() . . 60
Funkcje sieciowe gniazd 61
Obsługa systemu nazw domen (DNS) 61
Zgodność z Berkeley Software Distributions (BSD) . . 63
Zgodność z UNIX 98 65
Przekazywanie deskryptorów pomiędzy procesami – funkcje sendmsg() i recvmsg() 68
Scenariusze dla gniazd: tworzenie aplikacji komunikujących się z klientami IPv4 i IPv6 69
Przykład: akceptowanie połączeń od klientów IPv6 i IPv4 71
Przykład: klient IPv4 lub IPv6 75
Zalecenia dotyczące projektowania aplikacji używających gniazd 78
Przykłady: projekty aplikacji używających gniazd . . . 81
Przykłady: projekty aplikacji zorientowanych na połączenie 81
Przykład: korzystanie z asynchronicznych operacji we/wy 104
Przykłady: nawiązywanie chronionych połączeń . . 110
Przykład: procedury sieciowe obsługujące ochronę wątków i używające funkcji gethostbyaddr_r() . . . 144
Przykład: nieblokujące operacje we/wy i funkcja select() 145
Przykład: używanie sygnałów z blokującymi funkcjami API gniazd 151
Przykłady: użycie rozsyłania grupowego 154
Przykład: odpytywanie i aktualizacja serwera DNS 159
Przykład: przesyłanie danych za pomocą funkcji send_file() i accept_and_recv() 162
Narzędzie Xsockets 169
Konfigurowanie Xsockets 170
Konfigurowanie Xsockets do korzystania z przeglądarki WWW 173
Używanie Xsockets 176
Usuwanie obiektów utworzonych przez narzędzie Xsocket 177
Dostosowywanie Xsockets 178
Narzędzia do serwisowania. 178
Informacje pokrewne 179
Informacje dotyczące kodu 180

Uwagi. 181
Znaki towarowe 183
Warunki pobierania i drukowania publikacji 183

© Copyright IBM Corp. 2001, 2005

Programowanie z użyciem gniazd

Gniazdo to miejsce połączenia transmisji (punkt końcowy), które można nazwać i zaadresować w sieci. Procesy korzystające z gniazd mogą współistnieć w tym samym systemie, jak i w różnych systemach w różnych sieciach. Gniazda są przydatne zarówno dla aplikacji samodzielnych, jak i sieciowych. Gniazda umożliwiają wymianę informacji pomiędzy procesami na tej samej maszynie lub poprzez sieć, dystrybucję zadań do najbardziej wydajnej maszyny oraz ułatwiają dostęp do danych przechowywanych w centralnym miejscu. Funkcje API (application program interface) gniazd są standardem dla sieci TCP/IP. Funkcje te są obsługiwane w wielu systemach operacyjnych. Gniazda w systemie OS/400 obsługują wiele protokołów transportowych i sieciowych. Funkcje systemowe i funkcje sieciowe gniazd realizują ochronę wątków.

Programowanie z użyciem gniazd przedstawia wykorzystanie funkcji API gniazd do ustanowienia połączenia między procesem zdalnym i lokalnym. Programiści pracujący w środowisku Integrated Language Environment (ILE) C mogą skorzystać z niniejszych informacji przy tworzeniu aplikacji używających gniazd. Programowanie z wykorzystaniem funkcji API gniazd jest także możliwe dla innych języków w środowisku ILE, na przykład dla RPG. Więcej informacji o środowisku ILE RPG zawiera dokumentacja techniczna IBM Who Knew You Could Do That with RPG IV? A Sorcerer's Guide to System Access and More.

Również język Java obsługuje interfejs programowania z użyciem gniazd. Szczegóły zawiera temat Java w Centrum informacyjnym.

Sekcje dotyczące programowania z użyciem gniazd:

Wymienione niżej sekcje zawierają idee, zalecenia projektowe i przykłady pomocne przy tworzeniu aplikacji używających gniazd:

- **Drukowanie tego dokumentu**
  Przeglądanie i drukowanie publikacji Programowanie z użyciem gniazd w wersji PDF.
- **Wymagania wstępne dla programowania z użyciem gniazd**
  Zadania, które trzeba wykonać przed rozpoczęciem pisania aplikacji z użyciem funkcji API gniazd.
- **Podstawy projektowania gniazd**
  Przegląd przykładowych programów dla większości podstawowych typów gniazd. Dostępne poprzez odsyłacze programy przykładowe ilustrują podstawowe strategie projektowania gniazd.
- **Pojęcia dotyczące gniazd**
  Zaawansowane koncepcje dotyczące gniazd, na przykład asynchroniczne operacje we/wy i Global Secure Toolkit (GSKit). Odsyłacze zamieszczone w sekcji pozwalają przyjrzeć się przykładowym programom związanym z tymi koncepcjami.
- **Scenariusze dla gniazd: tworzenie aplikacji komunikujących się z klientami IPv4 i IPv6**
  Opis typowej sytuacji, w której użytkownik może użyć rodziny adresów AF_INET6. Od wersji V5R2 ta rodzina adresów zapewnia obsługę protokołu IP (Internet Protocol) w wersji 6 (IPv6). Protokół IPv6 obsługuje 128–bitowe adresy IP. W sekcji tej znajdują się również informacje dotyczące planowania i odsyłacze do przykładowych programów, które można zastosować w aplikacjach korzystających z rodziny adresów AF_INET6.
- **Zalecenia dotyczące projektowania aplikacji używających gniazd**
  Zagadnienia związane z projektowaniem efektywniejszych aplikacji używających gniazd.
- **Przykłady: projekty aplikacji używających gniazd**
  Przykłady programów używających gniazd, które można wykorzystać do tworzenia aplikacji używających gniazd.

  Uwaga: Znajduje się tu przykładowy kod programu. Sekcja Informacje dotyczące kodu zawiera szczegóły dotyczące użycia tych programów przykładowych.

© Copyright IBM Corp. 2001, 2005

- **Narzędzie Xsockets**
  Opis narzędzia Xsockets, które może być pomocne przy tworzeniu aplikacji używających gniazd. Są tu także odsyłacze do instrukcji instalowania i obsługi tego narzędzia.
- **Narzędzia do serwisowania**
  Opis narzędzia obsługi serwisowej dla gniazd.
- **Informacje pokrewne**
  Odsyłacze do innych źródeł informacji dotyczących gniazd i ich opisy.

Drukowanie tego dokumentu

W celu przeglądania i drukowania tego dokumentu można pobrać jego wersję PDF. Pliki PDF można przeglądać za pomocą programu Adobe® Acrobat® Reader. Jego kopię można pobrać ze strony

http://www.adobe.com/prodindex/acrobat/readstep.html .

Aby przejrzeć lub pobrać wersję PDF, wybierz Programowanie z użyciem gniazd (444 kB lub 132 strony).

Aby zapisać plik PDF na stacji roboczej w celu przeglądania lub drukowania:

1. Otwórz plik PDF w swojej przeglądarce (kliknij powyższy odsyłacz).
2. W menu przeglądarki kliknij **Plik**.
3. Kliknij **Zapisz jako...**
4. Przejdź do katalogu, w którym chcesz zapisać plik PDF.
5. Kliknij **Zapisz**.

Wymagania wstępne dla programowania z użyciem gniazd

Przed przystąpieniem do pisania aplikacji używającej gniazd wykonaj następujące zadania:

Wymagania dotyczące kompilatora

1. Zainstaluj bibliotekę QSYSINC. Biblioteka ta zawiera pliki nagłówkowe, niezbędne przy kompilowaniu aplikacji używających gniazd.
2. Zainstaluj program licencjonowany C Compiler (5722–CX2).

Wymagania dotyczące rodzin adresów AF_INET i AF_INET6

Oprócz spełnienia wymagań dotyczących kompilatora wykonaj następujące czynności:

1. Zaplanuj konfigurację protokołu TCP/IP.
2. Zainstaluj protokół TCP/IP.
3. Skonfiguruj protokół TCP/IP po raz pierwszy.
4. Skonfiguruj protokół TCP/IP do obsługi protokołu IPv6. Czynność ta jest opcjonalna. Interfejs IPv6 do protokołu TCP/IP należy skonfigurować wtedy, gdy użytkownik zamierza pisać aplikacje korzystające z rodziny adresów AF_INET6.

Wymagania dotyczące warstwy SSL (Secure Sockets Layer) i funkcji API Global Secure Toolkit (GSKit)

Aby pracować z warstwą SSL, oprócz spełnienia wymagań dotyczących kompilatora i adresów AF_INET oraz AF_INET6, wykonaj następujące czynności:

1. Zainstaluj i skonfiguruj program licencjonowany Menedżer certyfikatów cyfrowych (5722–SS1 opcja 34). Szczegóły znajdują się w sekcji Zarządzanie certyfikatami cyfrowymi w Centrum informacyjnym.
2. Zainstaluj program licencjonowany Cryptographic Access Provider (5722–AC3).
3. Aby używać warstwy SSL ze sprzętem szyfrującym, zainstaluj i skonfiguruj produkt 2058 Cryptographic Accelerator for iSeries lub 4758 PCI Cryptographic Coprocessor for iSeries. Produkt 2058 Cryptographic Accelerator umożliwia przeniesienie obciążenia związanego z szyfrowaniem warstwy SSL z systemu operacyjnego

na kartę. Pełny opis produktu 2058 Cryptographic Accelerator i jego funkcji zawiera sekcja 2058 Cryptographic Accelerator for iSeries. Produkt 4758 Cryptographic Coprocessor może służyć do szyfrowania warstwy SSL, jednakże w przeciwieństwie do produktu 2058 udostępnia więcej funkcji szyfrujących, na przykład szyfrowanie i deszyfrowanie kluczy. Opis funkcji i instrukcje dotyczące konfigurowania tej karty zawiera sekcja 4758 PCI Cryptographic Coprocessor for iSeries.

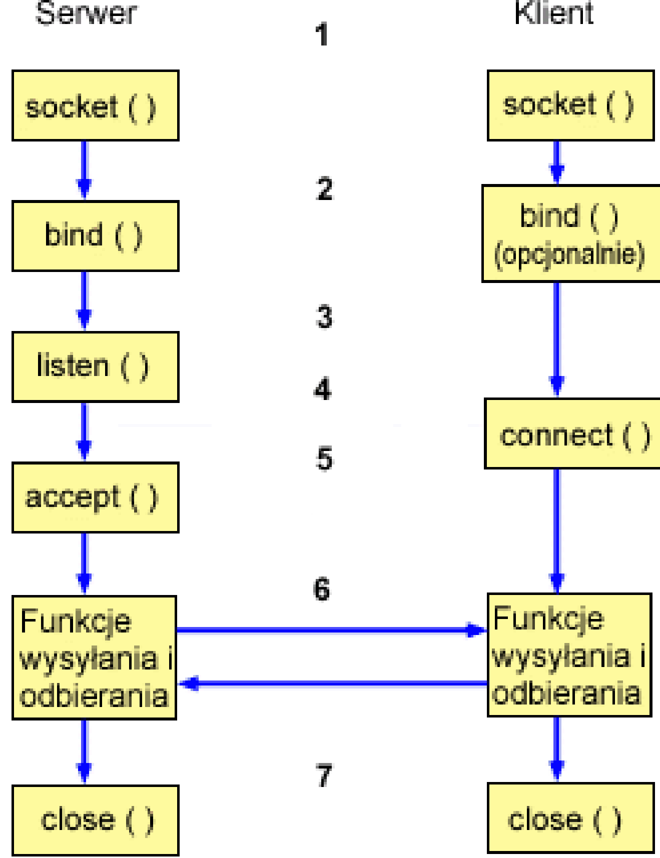
Jak działają gniazda

Gniazda są powszechnie używane w interakcjach klient/serwer. Typowa konfiguracja systemu umieszcza serwer na jednym komputerze, a klientów na innych komputerach. Klienci łączą się z serwerem, wymieniają informacje, a następnie się odłączają.

Ustanawianie gniazd wiąże się z typowym przebiegiem zdarzeń. W zorientowanym na połączenie modelu klient-serwer gniazdo procesu serwera czeka na żądania od klienta. W tym celu serwer najpierw ustanawia (wiąże) adres, z którego klient może skorzystać, aby znaleźć serwer. Gdy adres jest ustanowiony, wtedy serwer czeka, aż klient zażąda usługi. Wymiana danych od klienta do serwera zachodzi wtedy, gdy klient łączy się z serwerem poprzez gniazdo. Serwer spełnia żądanie klienta i wysyła odpowiedź z powrotem do klienta.

Uwaga: Obecnie IBM obsługuje dwie wersje większości funkcji API gniazd. Domyślne gniazda systemu OS/400 używają struktury i składni BSD (Berkeley Socket Distribution) 4.3. Różnice pomiędzy podstawowymi gniazdami systemu OS/400 a BSD 4.3 są opisane w sekcji Zgodność z BSD (Berkeley Socket Distribution). Druga wersja gniazd używa składni i struktur zgodnych z 4.4 i specyfikacją interfejsu programistycznego UNIX 98. Aby korzystać z interfejsu zgodnego ze standardem UNIX98, programiści mogą określić makrodefinicję _XOPEN_SOURCE. Opis funkcji API i różnic strukturalnych zawiera sekcja Zgodność z UNIX 98.

Na rysunku przedstawiono typowy przebieg zdarzeń (i sekwencję wywoływanych funkcji) sesji tworzenia gniazda zorientowanego na połączenie. Wyjaśnienie każdego zdarzenia znajduje się dalej.

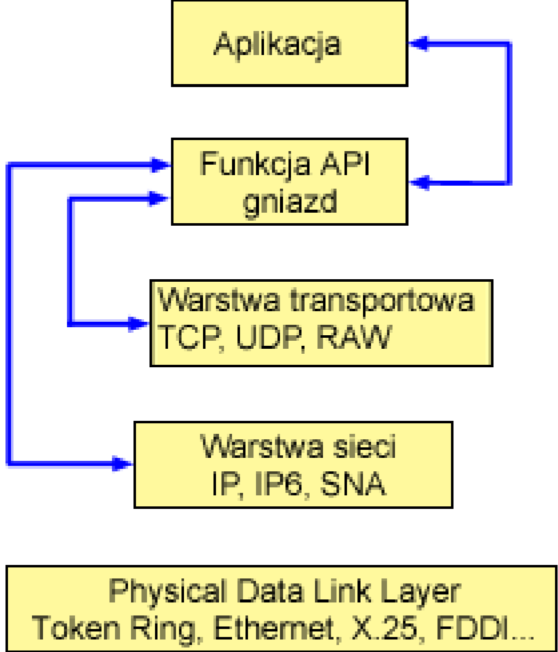


Typowy przebieg zdarzeń dla gniazd zorientowanych na połączenie:

1. Funkcja **socket()** tworzy punkt końcowy komunikacji i zwraca deskryptor gniazda reprezentujący ten punkt.
2. Gdy aplikacja ma deskryptor gniazda, może powiązać z gniazdem unikalną nazwę. Serwery muszą mieć powiązaną nazwę, aby były dostępne z sieci.
3. Funkcja **listen()** wskazuje gotowość do zaakceptowania wysyłanych przez klienta żądań połączenia. Gdy dla gniazda wywołana jest funkcja **listen()**, gniazdo nie może aktywnie zainicjować żądań połączenia. Funkcja API **listen()** jest wywoływana po przydzieleniu gniazda za pomocą funkcji **socket()** i po tym, jak funkcja **bind()** powiąże z gniazdem nazwę. Funkcję **listen()** trzeba wywoływać przed wywołaniem funkcji **accept()**.
4. Aplikacja typu klient korzysta z funkcji **connect()** w gnieździe potokowym do ustanowienia połączenia z serwerem.
5. Aplikacja typu serwer używa funkcji **accept()** do zaakceptowania żądania połączenia od klienta. Przed wywołaniem funkcji **accept()** na serwerze muszą być pomyślnie wywołane funkcje **bind()** i **listen()**.
6. Gdy pomiędzy gniazdami potokowymi (pomiędzy klientem a serwerem) jest ustanowione połączenie, można korzystać z dowolnej funkcji API gniazda służącej do przesyłania danych. Aplikacje typu klient i serwer mają do wyboru wiele funkcji przesyłania danych, takich jak **send()**, **recv()**, **read()**, **write()** i inne.
7. Gdy jedna ze stron chce zakończyć działanie, wywołuje funkcję **close()**, która zwalnia wszystkie przydzielone gniazdu zasoby systemu.

Uwaga:

Funkcje API gniazd w modelu komunikacyjnym znajdują się pomiędzy warstwą aplikacji a warstwą transportową. Nie stanowią one warstwy w modelu komunikacyjnym. Umożliwiają interakcję aplikacji z warstwami: transportową i sieciową w typowym modelu komunikacyjnym. Strzałki na poniższym rysunku pokazują pozycje gniazda i warstwę komunikacyjną realizowaną przez gniazdo.



Zazwyczaj konfiguracja sieci uniemożliwia połączenia między chronioną siecią wewnętrzną a mniej chronioną siecią zewnętrzną. Jednak można umożliwić gniazdom komunikację z programami typu serwer działającymi w systemie poza firewall (bardzo chroniony host).

Gniazda są również częścią implementacji IBM AnyNet dla architektury Multiprotocol Transport Networking (MPTN). Architektura MPTN umożliwia działanie sieci transportowej ponad dodatkowymi sieciami transportowymi i łączenie programów użytkowych przez sieci transportowe różnych typów.

Centrum informacyjne daje dostęp do informacji o funkcjach API na kilka sposobów. Temat Funkcje API gniazd zawiera przegląd funkcji i struktur gniazd. Aby znaleźć określoną funkcję API lub przeprowadzić wyszukiwanie w kategoriach funkcji API, dostępny jest interaktywny program do wyszukiwania funkcji API.

Charakterystyki gniazd

Gniazda mają następujące cechy:

- Gniazdo jest reprezentowane przez liczbę całkowitą. Liczbę tę nazywa się **deskryptorem gniazda**.
- Gniazdo istnieje tak długo, jak długo proces utrzymuje otwarte odniesienie do niego.
- Gniazdo można nazwać i wykorzystywać do komunikacji z innymi gniazdami w domenie komunikacyjnej.
- Gniazda komunikują się, gdy serwer przyjmuje od nich połączenia lub gdy wymieniają komunikaty z serwerem.
- Gniazda można tworzyć parami (tylko dla gniazd w rodzinie adresów AF_UNIX).

Połączenie zapewniane przez gniazdo może być zorientowane na połączenie lub bezpołączeniowe. Komunikacja **zorientowana na połączenie** zakłada, że ustanawiane jest połączenie i następuje dialog pomiędzy programami. Program realizujący usługę (program serwera) uruchamia dostępne gniazdo, które może akceptować przychodzące żądania połączeń. Opcjonalnie serwer może przypisać udostępnianej usłudze nazwę, która umożliwi klientom identyfikację miejsca, w którym ta usługa jest dostępna, i sposobu połączenia się z nią. Klient usługi (program typu klient) musi zażądać usługi od programu typu serwer. Klient realizuje to przez połączenie się z niepowtarzalną nazwą lub z atrybutami powiązanymi z tą nazwą, wyznaczonymi przez serwer. Przypomina to wybieranie numeru telefonu (identyfikator) i nawiązywanie połączenia z inną firmą, która oferuje usługę (na przykład hydrauliczną). Gdy odbiorca wywołania (serwer, a w tym przykładzie firma hydrauliczna) odbierze telefon, połączenie zostaje ustanowione. Hydraulik potwierdza, że jest tą firmą, o którą chodziło, i połączenie zostaje aktywne tak długo, na ile to potrzebne.

Komunikacja **bezpołączeniowa** zakłada, że nie zostaje ustanowione żadne połączenie, przez które będzie odbywał się dialog lub przesyłanie danych. Zamiast tego program typu serwer wyznacza nazwę identyfikującą miejsce, w którym jest osiągalny (coś jak skrzynka pocztowa). Wysyłając list do skrzynki pocztowej nie można być całkowicie pewnym, że dotrze on do odbiorcy. Zwykle trzeba czekać na odpowiedź. W takim przypadku nie ma aktywnego połączenia w czasie rzeczywistym, podczas którego wymieniane są dane.

Określanie charakterystyk gniazd

Gdy aplikacja tworzy gniazdo za pomocą funkcji **socket()**, musi zidentyfikować gniazdo przez podanie następujących parametrów:

- Rodzina adresów gniazda określa format struktury adresu gniazda. W tej sekcji przedstawiono przykłady struktury adresu dla każdej rodziny adresów. Ogólną definicję struktury adresów gniazd można znaleźć w sekcji Struktura adresu gniazda.
- Typ gniazda określa pożądaną formę komunikacji dla gniazda.
- Protokoły obsługiwane przez gniazdo to protokoły, których używa gniazdo.

Te parametry czy charakterystyki definiują aplikację używającą gniazd i sposób, w jaki współdziała ona z innymi aplikacjami używającymi gniazd. W zależności od rodziny adresów używanej przez gniazdo, można wybrać różne typy gniazd i różne protokoły. Poniższa tabela przedstawia rodziny adresów i powiązane z nimi typy gniazd i protokoły:

Tabela 1. Podsumowanie charakterystyk gniazd

Rodzina adresów	Typ gniazda	Protokół gniazda
AF_UNIX	SOCK_STREAM	nie dotyczy
	SOCK_DGRAM	nie dotyczy
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	nie dotyczy
AF_UNIX_CCSID	SOCK_STREAM	nie dotyczy
	SOCK_DGRAM	nie dotyczy

Oprócz tych parametrów gniazd, w procedurach sieciowych i plikach nagłówkowych dostarczanych z biblioteką QSYSINC są zdefiniowane stałe wartości. Opisy plików nagłówkowych można znaleźć w wykazie poszczególnych funkcji API znajdującym się w sekcji Funkcje API gniazd w Centrum informacyjnym. W sekcjach zawierających opis oraz składnię i sposób użycia każdej funkcji API wymieniono odpowiedni dla niej plik nagłówkowy.

Procedury obsługi gniazd w sieci umożliwiają aplikacjom używającym gniazd uzyskiwanie informacji z serwerów DNS, hostów, protokołów, usług oraz plików sieciowych. Opis tych procedur zawiera sekcja Procedury obsługi gniazd w sieci.

Struktura adresu gniazda

Podczas przekazywania i odbierania adresów gniazda korzystają ze struktury adresu **sockaddr**. Struktura ta nie wymaga API gniazda do rozpoznawania formatu adresowania. Obecnie system OS/400 obsługuje gniazda Berkeley Software Distributions (BSD) 4.3 i X/Open Single Unix Specification (UNIX 98). Podstawowe funkcje API systemu OS/400 obsługują struktury i składnię BSD 4.3. Definiując wartość 520 lub większą dla makra _XOPEN_SOURCE, można wybrać interfejs zgodny ze standardem UNIX 98. Dla każdej użytej struktury gniazd w standardzie BSD 4.3 będzie istniał odpowiednik w strukturze UNIX 98.

Tabela 2. Porównanie struktur adresowych gniazd BSD 4.3 i UNIX 98/BSD 4.4

Struktura BSD 4.3	Struktura zgodna z BSD 4.4/UNIX 98

`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];`
	`};`

Tabela 3. Struktura adresów

Pole struktury adresu	Definicja
sa_len	Pole to zawiera długość adresu w specyfikacji UNIX 98. **Uwaga:** Pole sa_len służy wyłącznie do zapewnienia zgodności ze standardem BSD 4.4. Nie ma potrzeby używania go, nawet w przypadku korzystania ze zgodności ze standardami BSD 4.4/UNIX 98. Pole jest ignorowane w przypadku adresów wejściowych.
sa_family	Pole to definiuje rodzinę adresów. Wartość ta jest podawana dla rodziny adresów w wywołaniu funkcji **socket()**.
sa_data	Pole to zawiera 14 bajtów zarezerwowanych do przechowywania samego adresu. **Uwaga:** Łańcuch znaków pola sa_data o długości 14 bajtów jest przeznaczony na adres. Adres może przekroczyć tę długość. Struktura jest ogólna, gdyż nie definiuje formatu adresu. Format adresu jest określony typem transportu, dla którego zostało utworzone gniazdo. Każdy z protokołów warstwy transportowej definiuje dokładny format, odpowiadający jego wymaganiom, w podobnej strukturze adresu. Protokół transportowy jest identyfikowany przez wartość parametru protokołu dla funkcji API **socket()**.
sockaddr_storage	Deklaruje obszar pamięci dla dowolnego adresu z tej rodziny adresów. Struktura ta jest wystarczająco duża i dopasowana do wszystkich struktur zależnych od użytego protokołu. Można zatem podczas korzystania z funkcji API rzutować ją jako strukturę as_sockaddr. Pole ss_family struktury sockaddr_storage zawsze będzie dopasowane do pola rodziny wszystkich struktur zależnych od protokołu.

Rodzina adresów gniazd

Parametr rodziny adresów dla funkcji **socket()** określa format struktury adresu, która będzie używana przez funkcje gniazd. Protokoły rodziny adresów zapewniają transport sieciowy danych aplikacji z jednej aplikacji do innej (lub z jednego procesu do innego wewnątrz tej samej maszyny). Aplikacja określa protokół transportu sieciowego w parametrze protokołu gniazda.

Parametr rodziny adresów (address_family) dla funkcji **socket()** określa strukturę adresu używaną przez funkcje gniazd. W poniższych sekcjach opisano wszystkie opisane wyżej rodziny adresów, ich zastosowanie, powiązane z nimi protokoły i przykłady odpowiednich struktur:

- Rodzina adresów AF_INET

- Rodzina adresów AF_INET6
- Rodzina adresów AF_UNIX
- Rodzina adresów AF_UNIX_CCSID
- Rodzina adresów AF_TELEPHONY

Rodzina adresów AF_INET

Umożliwia komunikację międzyprocesową pomiędzy procesami działającymi w tym samym systemie lub w różnych systemach. Adresy dla gniazd AF_INET to adresy IP i numery portów. Adres dla gniazda AF_INET można podać w postaci adresu IP, na przykład 130.99.128.1, lub w formie 32-bitowej, czyli X'82638001'.

Dla aplikacji używającej gniazd i korzystającej z protokołu IP wersja 4 (IPv4), rodzina adresów AF_INET używa struktury adresu **sockaddr_in**. Po zastosowaniu makra _XOPEN_SOURCE struktura adresów AF_INET ulega zmianie i staje się zgodna ze specyfikacjami BSD 4.4/UNIX 98. W przypadku struktury adresów sockaddr_in różnice te podaje poniższa tabela:

Tabela 4. Różnice w strukturach adresów sockaddr_in pomiędzy specyfikacjami BSD 4.3 a BSD 4.4/UNIX 98

Struktura adresów sockaddr_in BSD 4.3	Struktura adresów sockaddr_in BSD 4.4/UNIX 98

`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];`
	`};`

Tabela 5. Struktura adresów AF_INET

Pole struktury adresu	Definicja
sin_len	Pole to zawiera długość adresu w specyfikacji UNIX 98. **Uwaga:** Pole sin_len służy wyłącznie do zapewnienia zgodności ze standardem BSD 4.4. Nie ma potrzeby używania go, nawet w przypadku korzystania ze zgodności ze standardami BSD 4.4/UNIX 98. Pole jest ignorowane w przypadku adresów wejściowych.
sin_family	Rodzina adresów; w przypadku TCP lub UDP jest nią zawsze AF_INET.
sin_port	Numer portu.
sin_addr	Adres internetowy.
sin_zero	Pole zastrzeżone. W pole należy wpisać szesnastkowe zera.

Informacje o zastosowaniu rodziny adresów AF_INET i przykłady programów używających tej rodziny zawiera sekcja Korzystanie z rodziny adresów AF_INET.

Rodzina adresów AF_INET6

Ta rodzina adresów zapewnia obsługę protokołu IP (Internet Protocol) w wersji 6 (IPv6). Rodzina adresów AF_INET6 używa adresów 128-bitowych (16-bajtowych). W uproszczeniu, architektura tych adresów obejmuje 64 bity numeru sieci i 64 bity numeru hosta. Adresy z rodziny AF_INET6 można podawać w postaci x:x:x:x:x:x:x:x, gdzie 'x' oznacza szesnastkowe wartości ośmiu 16–bitowych części adresu. Przykładowo, adres może wyglądać następująco: FEDC:BA98:7654:3210:FEDC:BA98:7654:3210.

Dla aplikacji używającej gniazd i korzystającej z protokołów TCP, UDP lub RAW, rodzina adresów AF_INET6 używa struktury adresu **sockaddr_in6**. Struktura ta ulegnie zmianie, gdy do zapewnienia zgodności ze specyfikacjami BSD 4.4/UNIX 98 zostanie zastosowana makrodefinicja _XOPEN_SOURCE. W przypadku struktury adresów sockaddr_in6 różnice te podaje poniższa tabela:

Tabela 6. Różnice w strukturach adresów sockaddr_in6 pomiędzy specyfikacjami BSD 4.3 a BSD 4.4/UNIX 98

Struktura adresów sockaddr_in6 BSD 4.3	Struktura adresów sockaddr_in6 BSD 4.4/UNIX 98

`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;`
	`};`

Tabela 7. Struktura adresów AF_INET6

Pole struktury adresu	Definicja
sin6_len	Pole to zawiera długość adresu w specyfikacji UNIX 98. **Uwaga:** Pole sin6_len służy wyłącznie do zapewnienia zgodności ze standardem BSD 4.4. Nie ma potrzeby używania go, nawet w przypadku korzystania ze zgodności ze standardami BSD 4.4/UNIX 98. Pole jest ignorowane w przypadku adresów wejściowych.
sin6_family	Pole to określa, że zostanie użyta rodzina adresów AF_INET6.
sin6_port	Pole to zawiera port warstwy protokołu transportowego.
sin6_flowinfo	Pole to zawiera dwie informacje: klasę ruchu i etykietę przepływu. **Uwaga:** Nie jest ono obecnie obsługiwane i aby zapewnić zgodność z przyszłymi systemami, należy ustawić jego wartość na zero.
sin6_addr	Pole to określa adres IPv6.
sin6_scope_id	Pole to identyfikuje zestaw interfejsów odpowiednich dla zakresu adresów określonych w polu **sin6_addr**. **Uwaga:** Nie jest ono obecnie obsługiwane i aby zapewnić zgodność z przyszłymi systemami, należy ustawić jego wartość na zero.

Rodzina adresów AF_UNIX

Umożliwia komunikację międzyprocesową w ramach jednego systemu, w którym używane są funkcje API gniazd. Adres jest w rzeczywistości nazwą ścieżki do pozycji systemu plików. Gniazda można tworzyć w katalogu głównym lub w dowolnym otwartym systemie plików, na przykład asQSYS lub QDOC. Aby odbierać odsyłane datagramy, program musi powiązać gniazdo AF_UNIX, SOCK_DGRAM z nazwą. Dodatkowo, po zamknięciu gniazda, program musi w sposób jawny usunąć obiekt systemu plików funkcją API **unlink()**.

Gniazda w ramach rodziny adresów AF_UNIX korzystają ze struktury adresów **sockaddr_un**. Struktura ta ulegnie zmianie, gdy do zapewnienia zgodności ze specyfikacjami BSD 4.4/UNIX 98 zostanie zastosowana makrodefinicja _XOPEN_SOURCE. W przypadku struktury adresów sockaddr_un różnice te podaje poniższa tabela:

Tabela 8. Różnice w strukturach adresów sockaddr_un pomiędzy specyfikacjami BSD 4.3 a BSD 4.4/UNIX 98

Struktura adresów sockaddr_un BSD 4.3	Struktura adresów sockaddr_un BSD 4.4/UNIX 98

`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];`
	`};`

Tabela 9. Struktura adresu AF_UNIX

Pole struktury adresu	Definicja
sun_len	Pole to zawiera długość adresu w specyfikacji UNIX 98. **Uwaga:** Pole sun_len służy wyłącznie do zapewnienia zgodności ze standardem BSD 4.4. Nie ma potrzeby używania go, nawet w przypadku korzystania ze zgodności ze standardami BSD 4.4/UNIX 98. Pole jest ignorowane w przypadku adresów wejściowych.
sun_family	Rodzina adresów.
sun_path	Nazwa ścieżki do pozycji systemu plików.

Dla rodziny adresów AF_UNIX nie określa się protokołów ponieważ nie są one używane. Stosowany mechanizm komunikacji dwóch procesów zależy od danej maszyny.

Informacje o zastosowaniu rodziny adresów AF_UNIX i przykłady programów używających tej rodziny zawiera sekcja Korzystanie z rodziny adresów AF_UNIX.

Rodzina adresów AF_UNIX_CCSID

Jest zgodna z rodziną adresów AF_UNIX i ma takie same ograniczenia. Obie rodziny mogą być używane w komunikacji bezpołączeniowej lub zorientowanej na połączenie i żadne zewnętrzne funkcje komunikacyjne nie są używane do łączności między procesami. Różnica między rodzinami polega na tym, że rodzina adresów AF_UNIX_CCSID korzysta ze struktury adresu **sockaddr_unc**. Ta struktura adresu jest podobna do sockaddr_un, ale pozwala na nazwy ścieżek w kodzie UNICODE lub w dowolnym identyfikatorze CCSID, poprzez użycie formatu **Qlg_Path_Name_T**. Patrz format nazwy ścieżki w Centrum informacyjnym.

Ponieważ jednak gniazdo AF_UNIX może zwrócić nazwę ścieżki z gniazda AF_UNIX_CCSID w strukturze adresów AF_UNIX, wielkość ścieżki jest ograniczona. Rodzina AF_UNIX obsługuje tylko 126 znaków, więc rodzina AF_UNIX_CCSID jest również ograniczona do 126 znaków.

Użytkownik nie może wymieniać adresów AF_UNIX i AF_UNIX_CCSID w jednym gnieździe. Jeśli podczas wywołania funkcji **socket()** określi się rodzinę AF_UNIX_CCSID, wszystkie adresy w późniejszych wywołaniach funkcji API muszą być w strukturze **sockaddr_unc**.

```
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;
    } ;
```

Tabela 10. Struktura adresu AF_UNIX_CCSID

Pole struktury adresu	Definicja
sunc_family	Rodzina adresów; w tym przypadku zawsze AF_UNIX_CCSID.

Tabela 10. Struktura adresu AF_UNIX_CCSID (kontynuacja)

sunc_format	To pole zawiera dwie określone wartości dla formatu nazwy ścieżki: • **SO_UNC_DEFAULT** oznacza długą nazwę ścieżki z użyciem bieżącego identyfikatora CCSID dla nazw ścieżek zintegrowanego systemu plików. Pole sunc_qlg jest ignorowane. • **SO_UNC_USE_QLG** oznacza, że pole **sunc_qlg** definiuje format i identyfikator CCSID nazwy ścieżki.
sunc_zero	Pole zastrzeżone. W pole należy wpisać szesnastkowe zera.
sunc_qlg	Format nazwy ścieżki.
sunc_path	Nazwa ścieżki. Maksymalna długość nazwy wynosi 126 znaków i może być jedno- lub dwubajtowa. Nazwa może być zawarta w polu sunc_path field lub przydzielona osobno i wskazana poprzez wartość w polu sunc_path. Format nazwy jest określony wartościami pól sunc_format i sunc_qlg.

Więcej informacji o gniazdach AF_UNIX_CCSID oraz przykładowy program zawiera sekcja Korzystanie z rodziny adresów AF_UNIX_CCSID.

Rodzina adresów AF_TELEPHONY

Pozwala użytkownikom nawiązywać i odbierać połączenia telefoniczne poprzez sieć telefoniczną ISDN za pomocą standardowych funkcji API gniazd. Gniazda tworzące punkty końcowe połączenia w tej domenie są w rzeczywistości wywoływanymi i wywołującymi stronami połączenia telefonicznego. Adresy w tej rodzinie adresów są reprezentowane przez 40-cyfrowe numery telefoniczne. Ta rodzina adresów jest najczęściej wykorzystywana do obsługi faksów.

System obsługuje gniazda AF_TELEPHONY tylko jako gniazda zorientowane na połączenie, dla których typ gniazda to SOCK_STREAM. Połączenie z gniazdem w domenie telefonicznej nie jest bardziej niezawodne niż stanowiące jego podstawę połączenie telefoniczne. Jeśli potrzebna jest gwarancja dostarczenia, należy użyć aplikacji, które oferują takie usługi, na przykład aplikacji do obsługi faksów wykorzystujących tę rodzinę adresów.

Gniazda z rodziny adresów AF_TELEPHONY korzystają ze struktury adresu **sockaddr_tel**:

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

Adres telefoniczny składa się z 2 bajtów długości, po których następuje numer telefonu zawierający nie więcej niż 40 cyfr (0-9).

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

Tabela 11. Struktura adresu AF_TELEPHONY

Pole struktury adresu	Definicja
stel_family	Rodzina adresów.
stel_addr	Adres telefoniczny.
stel_zero	Pole zastrzeżone.

Więcej informacji o rodzinie adresów AF_TELEPHONY można znaleźć w sekcji Korzystanie z rodziny adresów AF_TELEPHONY, która zawiera również instrukcje konfigurowania środowiska do korzystania z rodziny adresów AF_TELEPHONY.

Typy gniazd

Drugi parametr w wywołaniu gniazda określa typ gniazda. Typ gniazda umożliwia identyfikację typu i charakterystyki połączenia, które zostanie nawiązane w celu transportu danych z jednej maszyny do innej lub z jednego procesu do innego. Poniższa lista opisuje typy gniazd obsługiwane przez serwer iSeries:

Strumieniowe (SOCK_STREAM)
Ten rodzaj gniazda jest zorientowany na połączenie. Ustanawia kompleksowe połączenie za pomocą funkcji **bind()**, **listen()**, **accept()** i **connect()**. SOCK_STREAM wysyła dane bez błędów czy powtórzeń i otrzymuje dane w kolejności wysyłania. SOCK_STREAM stosuje sterowanie przepływem, aby uniknąć przekroczenia granicy danych. Nie narzuca granic bloków dla danych. Zakłada, że dane stanowią strumień bajtów. W implementacji iSeries gniazd strumieniowych można używać w sieciach Transmission Control Protocol (TCP) i Systems Network Architecture (SNA), a także dla gniazd AF_UNIX, AF_UNIX_CCSID i AF TELEPHONY.Gniazd strumieniowych można także używać do połączeń się z systemami poza hostem chronionym (firewallem).

Datagramowe (SOCK_DGRAM)
W terminologii protokołu IP najprostszą jednostką przesyłania danych jest **datagram**. Jest to nagłówek, po którym następują pewne dane. Gniazdo datagramowe jest bezpołączeniowe. Nie nawiązuje żadnej kompleksowej komunikacji z protokołem transportowym. Gniazdo wysyła datagramy jako niezależne pakiety bez żadnej gwarancji dostarczenia. Dane mogą zaginąć lub zostać powielone. Datagramy mogą przybywać w dowolnej kolejności. Wielkość datagramu jest ograniczona do wielkości danych, które można wysłać w pojedynczej transakcji. W przypadku niektórych protokołów transportowych datagramy mogą korzystać z różnych tras przez sieć. Na tym rodzaju gniazda można wywołać funkcję **connect()**. Jednak w przypadku funkcji **connect()** należy określić adres docelowy, pod który program wysyła dane i spod którego je odbiera. W implementacji iSeries gniazd datagramowych można używać w sieciach z protokołami UDP i SNA oraz z rodzinami adresów AF_UNIX i AF_UNIX_CCSID.

Surowe (SOCK_RAW)
Ten rodzaj gniazda umożliwia bezpośredni dostęp do protokołów niższych warstw, takich jak IPv4 lub IPv6, oraz ICMP lub ICMP6. SOCK_RAW wymaga większego doświadczenia programistycznego, gdyż zarządza się informacjami z nagłówka protokołu używanymi przez protokół transportowy. Na tym poziomie protokół transportowy może narzucać format danych i semantykę, która jest dla niego specyficzna.

Protokoły gniazd

Protokoły umożliwiają sieciowy transport danych aplikacji z jednej maszyny do innej (lub z jednego procesu do innego). Aplikacja określa protokół transportowy w parametrze **protocol** funkcji **socket()**.

Używając rodziny adresów AF_INET można korzystać z kilku protokołów transportowych. Jednocześnie w danym gnieździe mogą być aktywne protokoły SNA, TCP/IP lub UDP/IP. Atrybut sieciowy ALWANYNET (Allow ANYNET support - Zezwolenie na obsługę AnyNet) umożliwia wybranie innego transportu niż TCP/IP dla aplikacji używających gniazd AF_INET. Atrybut ten może mieć wartość ***YES** lub ***NO**. Wartością domyślną jest ***NO**.

Jeśli na przykład bieżący (domyślny) status to *NO, gniazda AF_INET w sieci SNA nie są aktywne. Jeśli gniazda AF_INET mają być używane wyłącznie w sieci TCP/IP, atrybut ALWANYNET powinien mieć status *NO, aby efektywniej wykorzystać jednostkę centralną.

Uwaga: Atrybut sieciowy ALWANYNET ma również wpływ na komunikację APPC w sieciach TCP/IP.

Informacje o opcjach konfiguracyjnych APPC zawiera sekcja Konfigurowanie APPC, APPN i HPR.

Gniazda AF_INET w sieci TCP/IP mogą być również typu SOCK_RAW, co oznacza, że komunikują się one bezpośrednio z warstwą sieci zwaną Internet Protocol (IP). Zwykle z warstwą tą komunikują się protokoły TCP lub UDP. Przy korzystaniu z gniazd SOCK_RAW program użytkowy określa dowolny protokół z zakresu od 0 do 255 (z

wyjątkiem protokołów TCP i UDP). Gdy komputery komunikują się w sieci, ten numer protokołu jest następnie przesyłany w nagłówkach IP. W rezultacie program użytkowy pełni rolę protokołu transportowego, gdyż musi udostępnić wszystkie usługi, udostępniane normalnie przez protokoły UDP lub TCP.

W przypadku rodzin adresów AF_UNIX, AF_UNIX_CCSID i AF_TELEPHONY specyfikacja protokołów nie ma znaczenia, ponieważ rodziny te nie używają protokołów. Mechanizm komunikacji pomiędzy dwoma procesami na tym samym komputerze jest specyficzny dla danego komputera.

Podstawy projektowania gniazd

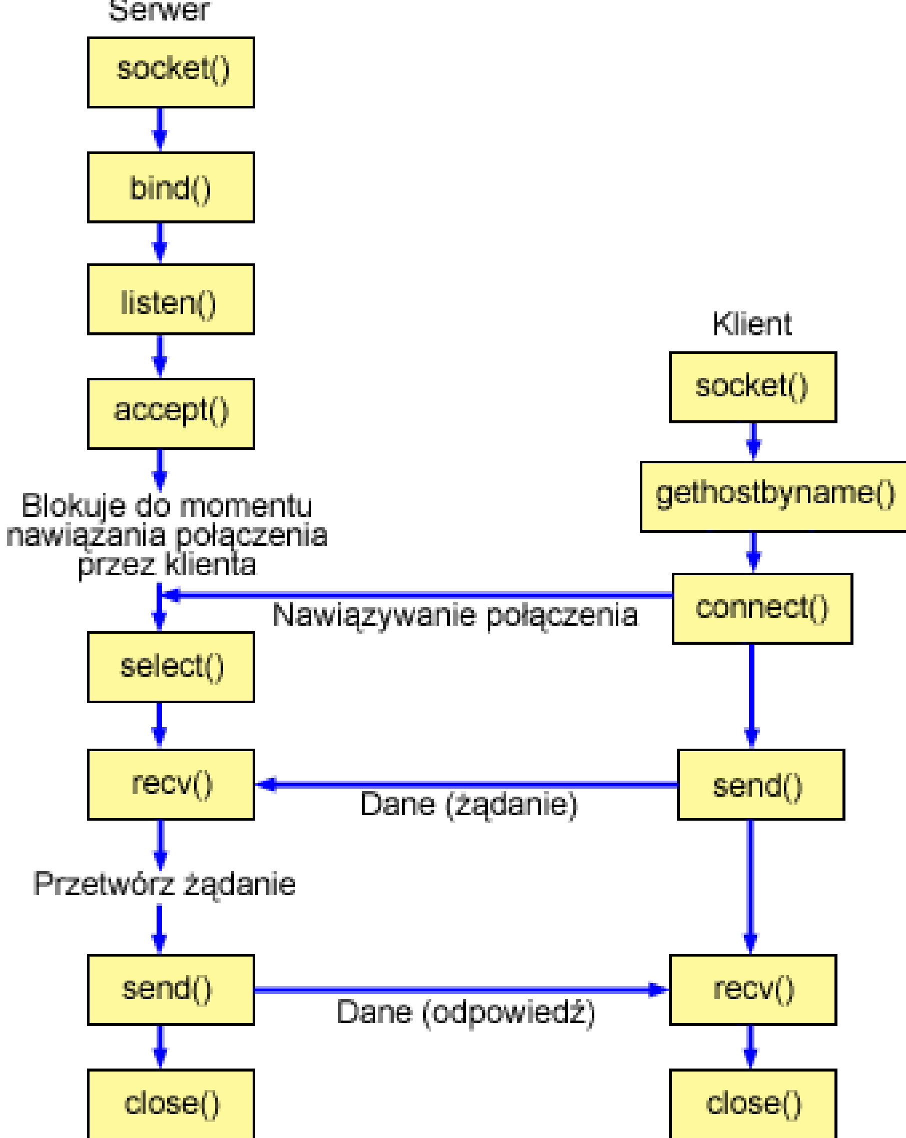
W tej sekcji przedstawiono przykłady programów, w których zaimplementowano najprostsze gniazda. Przykłady te stanowią podstawę dla konstrukcji bardziej zaawansowanych. Zaimplementowano w nich niektóre z podstawowych koncepcji opisanych w poprzednich sekcjach. Poniższe sekcje zawierają przykłady najpopularniejszych typów programów używających gniazd.

- Tworzenie gniazd zorientowanych na połączenie
- Tworzenie gniazd bezpołączeniowych
- Projektowanie aplikacji używających rodziny adresów

Tworzenie gniazd zorientowanych na połączenie

Niniejsze przykłady programów serwera i klienta ilustrują funkcje API gniazd napisane dla protokołu zorientowanego na połączenie, takiego jak TCP (Transmission Control Protocol).

Poniższy rysunek ilustruje relację klient/serwer funkcji API dla gniazd dla protokołu zorientowanego na połączenie.



Przebieg zdarzeń w gnieździe: serwer zorientowany na połączenie
Poniżej wyjaśniono sekwencję wywołań funkcji gniazd, przedstawionych na powyższej ilustracji. Opisano także relacje pomiędzy aplikacją serwera a aplikacją klienta w architekturze zorientowanej na połączenie. Każdy zbiór wywołań zawiera odsyłacze do uwag dotyczących użycia poszczególnych funkcji API. Aby uzyskać szczegółowe informacje dotyczące użycia tych funkcji API, należy użyć poniższych odsyłaczy. W sekcji Przykład: program serwera zorientowany na połączenie użyto wywołań następujących funkcji:

1. Funkcja **socket()** zwraca deskryptor gniazda odpowiadający punktowi końcowemu. Instrukcja ta informuje również, że dla tego gniazda zostanie użyta rodzina adresów INET (Internet Protocol) z transportem TCP transport (SOCK_STREAM).
2. Funkcja **setsockopt()** umożliwia ponowne użycie adresu lokalnego po restarcie serwera, zanim upłynie wymagany czas oczekiwania.
3. Po utworzeniu deskryptora gniazda funkcja **bind()** pobiera unikalną nazwę gniazda. W tym przykładzie użytkownik ustawia wartość s_addr na zero, co umożliwia nawiązanie połączenia z dowolnego klienta IPv4, który określi port 3005.

4. Funkcja **listen()** umożliwia serwerowi przyjęcie połączenia przychodzącego od klienta. W tym przykładzie kolejka (backlog) ma wartość 10. Oznacza to, że system umieści w kolejce pierwsze 10 przychodzących żądań połączenia, a następne odrzuci.
5. Serwer używa funkcji **accept()** do zaakceptowania żądania połączenia przychodzącego. Wywołanie funkcji **accept()** zostanie zablokowane na nieokreślony czas oczekiwania na połączenie przychodzące.
6. Funkcja **select()** powoduje, że proces oczekuje na wystąpienie zdarzenia, po którym kontynuuje działanie. W tym przykładzie system powiadamia proces dopiero wtedy, gdy dane do odczytania będą dostępne. W tym wywołaniu funkcji select() został określony limit czasu równy 30 sekund.
7. Funkcja **recv()** odbiera dane z aplikacji klienta. W tym przykładzie wiadomo, że klient wyśle 250 bajtów danych. Na tej podstawie można użyć opcji gniazda SO_RCVLOWAT i określić, że funkcja **recv()** ma pozostać w uśpieniu do momentu nadejścia całych 250 bajtów danych.
8. Funkcja **send()** odsyła dane do klienta.
9. Funkcja **close()** zamyka wszystkie otwarte deskryptory gniazd.

Przebieg zdarzeń w gnieździe: klient zorientowany na połączenie

W sekcji Przykład: program klienta zorientowany na połączenie użyto wywołań następujących funkcji:

1. Funkcja **socket()** zwraca deskryptor gniazda odpowiadający punktowi końcowemu. Instrukcja ta informuje również, że dla tego gniazda zostanie użyta rodzina adresów INET (Internet Protocol) z transportem TCP transport (SOCK_STREAM).
2. W przykładowym programie klienta, jeśli przesłany do funkcji **inet_addr()** ciąg znaków serwera nie jest adresem IP składającym się z liczb dziesiętnych oddzielonych kropkami, zakłada się, że jest to nazwa hosta serwera. W takim przypadku do pobrania adresu IP serwera używa się funkcji **gethostbyname()**.
3. Po odebraniu deskryptora gniazda należy użyć funkcji **connect()** do nawiązania połączenia z serwerem.
4. Funkcja **send()** wysyła 250 bajtów danych do serwera.
5. Funkcja **recv()** czeka na odesłanie 250 bajtów z serwera. W tym przykładzie wiadomo, że serwer odpowie, przesyłając te same 250 bajtów danych, które otrzyma z klienta. W przykładowym kliencie te 250 bajtów może dotrzeć w oddzielnych pakietach, dlatego używamy funkcji **recv()** wielokrotnie do momentu, gdy zostanie odebrane całe 250 bajtów.
6. Funkcja **close()** zamyka wszystkie otwarte deskryptory gniazd.

Przykład: program serwera zorientowany na połączenie

Poniższy przykładowy kod pokazuje, jak można utworzyć serwer zorientowany na połączenie. Na jego podstawie można utworzyć własną aplikację serwera gniazd. Serwer zorientowany na połączenie jest jednym z najczęściej używanych modeli aplikacji używających gniazd. W modelu zorientowanym na połączenie aplikacja serwera tworzy gniazdo, które służy do odbierania żądań z klientów. Informacje dotyczące wykorzystania przykładowych kodów zawiera sekcja Informacje dotyczące kodu.

```
/**************************************************************************/
/* Jest to przykładowy kod programu serwera zorientowanego na połączenie. */
/**************************************************************************/

/**************************************************************************/
/* Wymagane przez program pliki nagłówkowe.                               */
/**************************************************************************/
#include <stdio.h>
#include <sys/time.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

/**************************************************************************/
/* Stałe używane przez ten program                                        */
/**************************************************************************/
#define SERVER_PORT     3005
#define BUFFER_LENGTH    250
#define FALSE              0
```

```
void main()
{
   /*********************************************************************/
   /* Definicje zmiennych i struktur.                                   */
   /*********************************************************************/
   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;

   /*********************************************************************/
   /* Pętla do/while(FALSE) ułatwia realizację procedur czyszczących w  */
   /* przypadku błędu. Funkcja close() dla poszczególnych deskryptorów  */
   /* gniazd jest uruchamiana jednokrotnie na samym końcu programu.     */
   /*********************************************************************/
   do
   {
      /******************************************************************/
      /* Funkcja socket() zwraca deskryptor gniazda stanowiącego punkt  */
      /* końcowy. Instrukcja ta określa także, że dla tego gniazda      */
      /* użyta zostanie rodzina adresów INET (Internet Protocol)        */
      /* z protokołem transportowym TCP (SOCK_STREAM).                  */
      /******************************************************************/
      sd = socket(AF_INET, SOCK_STREAM, 0);
      if (sd < 0)
    {
         perror("Niepowodzenie socket()");
         break;
      }

      /******************************************************************/
      /* Funkcja setsockopt() umożliwia ponowne użycie adresu lokalnego  */
      /* przy ponownym uruchomieniu serwera, zanim upłynie wymagany czas */
      /* oczekiwania.                                                   */
      /******************************************************************/
      rc = setsockopt(sd, SOL_SOCKET, SO_REUSEADDR, (char *)&on, sizeof(on));
      if (rc < 0)
      {
         perror("Niepowodzenie setsockopt(SO_REUSEADDR)");
         break;
      }

      /******************************************************************/
      /* Po utworzeniu deskryptora gniazda funkcja bind() pobiera       */
      /* unikalną nazwę gniazda. W tym przykładzie użytkownik ustawia   */
      /* wartość s_addr na zero, co umożliwia nawiązanie połączenia z   */
      /* dowolnego klienta, który określi 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("Niepowodzenie bind()");
         break;
      }

      /******************************************************************/
      /* Funkcja listen() umożliwia serwerowi przyjęcie połączeń        */
      /* przychodzących od klienta. W tym przykładzie kolejka (backlog)  */
      /* ma wartość 10. Oznacza to, że system umieści w kolejce pierwsze */
      /* 10 przychodzących żądań połączenia, a następne będzie je       */
```

```
/* odrzucał.                                                     */
/*****************************************************************/
rc = listen(sd, 10);
if (rc < 0)
{
   perror("Niepowodzenie listen()");
   break;
}

printf("Gotowy do obsługi klienta (connect()).\n");

/*****************************************************************/
/* Serwer używa funkcji accept() do zaakceptowania połączenia      */
/* przychodzącego. Wywołanie funkcji accept() zostanie zablokowane  */
/* na nieokreślony czas oczekiwania na połączenie przychodzące.     */
/*****************************************************************/
sd2 = accept(sd, NULL, NULL);
if (sd2 < 0)
{
   perror("Niepowodzenie accept()");
   break;
}

/*****************************************************************/
/* Funkcja select() powoduje, że proces oczekuje na wystąpienie     */
/* zdarzenia, po którym kontynuuje działanie. W tym przykładzie     */
/* system powiadamia proces tylko wtedy, gdy dane do odczytania     */
/* będą dostępne. W tym wywołaniu funkcji select() został określony */
/* limit czasu równy 30 sekund.                                     */
/*****************************************************************/
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("Niepowodzenie select()");
   break;
 }

if (rc == 0)
{
   printf("Przekroczenie czasu dla select().\n");
   break;
}

/*****************************************************************/
/* W tym przykładzie wiadomo, że klient wyśle 250 bajtów danych.    */
/* Dzięki temu można użyć opcji gniazda SO_RCVLOWAT i określić, że  */
/* funkcja recv() ma nie wychodzić z uśpienia, dopóki nie zostanie  */
/* odebrane wszystkie 250 bajtów danych.                            */
/*****************************************************************/
length = BUFFER_LENGTH;
rc = setsockopt(sd2, SOL_SOCKET, SO_RCVLOWAT,
                                    (char *)&length, sizeof(length));
if (rc < 0)
{
   perror("Niepowodzenie setsockopt(SO_RCVLOWAT)");
   break;
}

/*****************************************************************/
/* Odebranie 250 bajtów od klienta                                  */
/*****************************************************************/
```

```
      rc = recv(sd2, buffer, sizeof(buffer), 0);
      if (rc < 0)
      {
         perror("Niepowodzenie recv()");
         break;
      }

      printf("Otrzymano dane, bajtów: %d\n", rc);
      if (rc == 0 ||
          rc < sizeof(buffer))
      {
         printf("Klient zamknął połączenie przed wysłaniem\n");
         printf("wszystkich danych\n");
         break;
      }

      /********************************************************************/
      /* Odesłanie danych do klienta                */
      /********************************************************************/
      rc = send(sd2, buffer, sizeof(buffer), 0);
      if (rc < 0)
      {
         perror("Niepowodzenie send()");
         break;
      }

      /********************************************************************/
      /* Zakończenie programu                                             */
      /********************************************************************/

   } while (FALSE);

   /***********************************************************************/
   /* Zamknięcie wszystkich otwartych deskryptorów gniazd                 */
   /***********************************************************************/
   if (sd != -1)
      close(sd);
   if (sd2 != -1)
      close(sd2);
}
```

Przykład: program klienta zorientowany na połączenie

Poniższy przykład ilustruje tworzenie programu klienta gniazd do łączenia z serwerem zorientowanym na połączenie. Klient usługi (program typu klient) musi zażądać usługi od programu typu serwer. W oparciu o ten przykładowy kod można pisać własne aplikacje typu klient. Informacje dotyczące wykorzystania przykładowych kodów zawiera sekcja Informacje dotyczące kodu.

```
/**************************************************************************/
/* Jest to przykładowy kod programu klienta zorientowanego na połączenie. */
/**************************************************************************/

/**************************************************************************/
/* Wymagane przez program pliki nagłówkowe.                               */
/**************************************************************************/
#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>

/**************************************************************************/
/* Stałe używane przez ten program                                        */
/**************************************************************************/
#define SERVER_PORT     3005
#define BUFFER_LENGTH    250
```

```
#define FALSE              0
#define SERVER_NAME     "NazwaSerweraHosta"

/* Przekaż 1 parametr, który albo jest     */
/* adresem, albo nazwą hosta serwera lub   */
/* ustaw nazwę serwera zmienną #define     */
/* SERVER_NAME.                            */
void main(int argc, char *argv[])
{
   /***********************************************************************/
   /* Definicje zmiennych i struktur.                                     */
   /***********************************************************************/
   int    sd=-1, rc, bytesReceived;
   char   buffer[BUFFER_LENGTH];
   char   server[NETDB_MAX_HOST_NAME_LENGTH];
   struct sockaddr_in serveraddr;
   struct hostent *hostp;

   /***********************************************************************/
   /* Pętla do/while(FALSE) ułatwia realizację procedur czyszczących w    */
   /* przypadku błędu. Funkcja close() dla deskryptora gniazda jest       */
   /* uruchamiana jednokrotnie na samym końcu programu.                   */
   /***********************************************************************/
   do
   {
      /********************************************************************/
      /* Funkcja socket() zwraca deskryptor gniazda stanowiącego punkt    */
      /* końcowy. Instrukcja ta określa także, że dla tego gniazda        */
      /* użyta zostanie rodzina adresów INET (Internet Protocol)          */
      /* z protokołem transportowym TCP (SOCK_STREAM).                    */
      /********************************************************************/
      sd = socket(AF_INET, SOCK_STREAM, 0);
      if (sd < 0)
    {
         perror("Niepowodzenie socket()");
         break;
      }

      /********************************************************************/
      /* Jeśli został przekazany argument, należy go użyć jako nazwy      */
      /* serwera, w przeciwnym razie należy użyć zmiennej określonej      */
      /* w makrze #define znajdującym się na początku programu.           */
      /********************************************************************/
      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)
      {
         /*****************************************************************/
         /* Łańcuch określający serwer, przekazany do funkcji inet_addr() */
         /* nie jest adresem IP składającym się z liczb dziesiętnych      */
         /* oddzielonych kropkami, zatem musi być to nazwa hosta serwera. */
         /* Do pobrania adresu IP serwera zostanie użyta funkcja          */
         /* gethostbyname().                                              */
         /*****************************************************************/

         hostp = gethostbyname(server);
         if (hostp == (struct hostent *)NULL)
         {
            printf("Hosta nie znaleziono --> ");
            printf("h_errno = %d\n", h_errno);
```

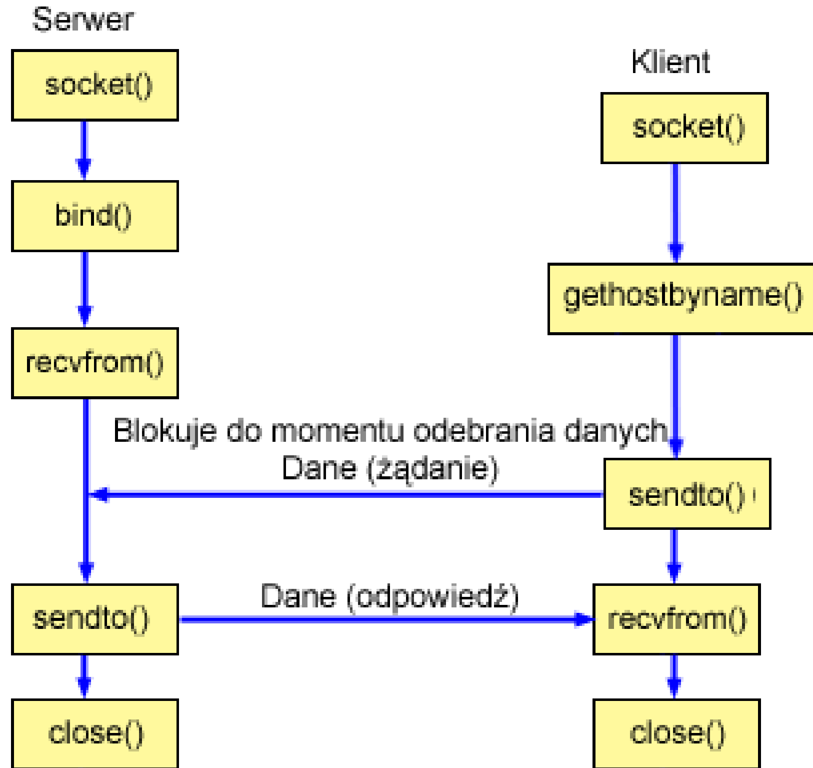
```
            break;
         }

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

      /********************************************************************/
      /* Aby nawiązać połączenie z serwerem, zostanie użyta funkcja       */
      /* connect().                                                       */
      /********************************************************************/
      rc = connect(sd, (struct sockaddr *)&serveraddr, sizeof(serveraddr));
      if (rc < 0)
      {
         perror("Niepowodzenie connect()");
         break;
      }

      /********************************************************************/
      /* Wysłanie 250 bajtów znaków 'a' do serwera                        */
      /********************************************************************/
      memset(buffer, 'a', sizeof(buffer));
      rc = send(sd, buffer, sizeof(buffer), 0);
      if (rc < 0)
      {
         perror("Niepowodzenie send()");
         break;
      }

      /********************************************************************/
      /* W tym przykładzie wiadomo, że serwer odpowie wysłaniem tych      */
      /* samych 250 bajtów, które wysłaliśmy. Ponieważ wiadomo, że        */
      /* zostanie odesłanych 250 bajtów, można użyć opcji gniazda         */
      /* SO_RCVLOWAT, uruchomić pojedynczą funkcję recv() i pobrać        */
      /* wszystkie dane.                                                  */
      /*                                                                  */
      /* Użycie opcji SO_RCVLOWAT zostało już pokazane w przykładzie      */
      /* serwera, dlatego tutaj użyjemy innej metody. Ponieważ te 250     */
      /* bajtów danych może być przysłanych w oddzielnych pakietach,      */
      /* będziemy wielokrotnie uruchamiali funkcję recv(), dopóki         */
      /* nie odbierzemy wszystkich 250 bajtów.                            */
      /********************************************************************/
      bytesReceived = 0;
      while (bytesReceived < BUFFER_LENGTH)
      {
         rc = recv(sd, & buffer[bytesReceived],
                   BUFFER_LENGTH - bytesReceived, 0);
         if (rc < 0)
         {
            perror("Niepowodzenie recv()");
            break;
         }
         else if (rc == 0)
         {
            printf("Serwer zamknął połączenie\n");
            break;
         }

         /*****************************************************************/
         /* Zwiększenie liczby otrzymanych dotychczas bajtów              */
         /*****************************************************************/
         bytesReceived += rc;
      }

   } while (FALSE);
```

```
   /**********************************************************************/
   /* Zamknięcie wszystkich otwartych deskryptorów gniazd                */
   /**********************************************************************/
   if (sd != -1)
      close(sd);
}
```

Tworzenie gniazd bezpołączeniowych

Gniazda bezpołączeniowe nie nawiązują połączenia, przez które dane mogłyby być przesyłane. Zamiast tego aplikacja serwera określa nazwę, do której klient może wysyłać żądania. Gniazda bezpołączeniowe zamiast protokołu TCP/IP używają protokołu UDP. Sekcje Przykład: tworzenie programu serwera bezpołączeniowego i Przykład: bezpołączeniowy program klienta przedstawiają funkcje API gniazd napisane dla protokołu UDP.

Poniższy rysunek ilustruje relację klient/serwer funkcji API gniazd, użytych w przykładowych programach, dla gniazda bezpołączeniowego.



Przebieg zdarzeń w gnieździe: serwer bezpołączeniowy
Poniżej wyjaśniono sekwencję wywołań funkcji gniazd, przedstawionych na powyższej ilustracji i w programach przykładowych. Opisano także relacje pomiędzy aplikacją serwera a aplikacją klienta w architekturze bezpołączeniowej. Każdy zbiór wywołań zawiera odsyłacze do uwag dotyczących użycia poszczególnych funkcji API. Aby uzyskać szczegółowe informacje dotyczące użycia tych funkcji API, należy użyć poniższych odsyłaczy. W sekcji Przykład: serwer bezpołączeniowy użyto wywołań następujących funkcji:

1. Funkcja **socket()** zwraca deskryptor gniazda odpowiadający punktowi końcowemu. Instrukcja ta informuje również, że dla tego gniazda zostanie użyta rodzina adresów INET (Internet Protocol) z transportem UDP transport (SOCK_DGRAM).
2. Po utworzeniu deskryptora gniazda funkcja **bind()** pobiera unikalną nazwę gniazda. W tym przykładzie użytkownik ustawia wartość s_addr na zero, co oznacza, że port UDP 3555 będzie powiązany z wszystkimi adresami IPv4 w systemie.
3. Do odebrania danych serwer użyje funkcji **recvfrom()**. Funkcja **recvfrom()** czeka na przesłanie danych przez czas nieokreślony.

4. Funkcja **sendto()** odsyła dane do klienta.
5. Funkcja **close()** zamyka wszystkie otwarte deskryptory gniazd.

Przebieg zdarzeń w gnieździe: klient bezpołączeniowy
W sekcji Przykład: klient bezpołączeniowy użyto wywołań następujących funkcji:

1. Funkcja **socket()** zwraca deskryptor gniazda odpowiadający punktowi końcowemu. Instrukcja ta informuje również, że dla tego gniazda zostanie użyta rodzina adresów INET (Internet Protocol) z transportem UDP transport (SOCK_DGRAM).
2. W przykładowym programie klienta, jeśli przesłany do funkcji inet_addr() ciąg znaków serwera nie jest adresem IP składającym się z liczb dziesiętnych oddzielonych kropkami, zakłada się, że jest to nazwa hosta serwera. W takim przypadku do pobrania adresu IP serwera używa się funkcji **gethostbyname()**.
3. Funkcja **sendto()** odsyła dane do serwera.
4. Do odebrania danych z powrotem z serwera zostanie użyta funkcja **recvfrom()**.
5. Funkcja **close()** zamyka wszystkie otwarte deskryptory gniazd.

Przykład: bezpołączeniowy program serwera

Tego przykładu można użyć do tworzenia własnego serwera bezpołączeniowego. Poniższy program bezpołączeniowego serwera gniazd używa protokołu UDP. Informacje dotyczące wykorzystania przykładowych kodów zawiera sekcja Informacje dotyczące kodu.

```
/**************************************************************************/
/* Jest to przykładowy kod programu serwera bezpołączeniowego.            */
/**************************************************************************/

/**************************************************************************/
/* Wymagane przez program pliki nagłówkowe.                               */
/**************************************************************************/
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

/**************************************************************************/
/* Stałe używane przez ten program                                        */
/**************************************************************************/
#define SERVER_PORT     3555
#define BUFFER_LENGTH    100
#define FALSE              0

void main()
{
   /***********************************************************************/
   /* Definicje zmiennych i struktur.                                     */
   /***********************************************************************/
   int    sd=-1, rc;
   char   buffer[BUFFER_LENGTH];
   struct sockaddr_in serveraddr;
   struct sockaddr_in clientaddr;
   int    clientaddrlen = sizeof(clientaddr);

   /***********************************************************************/
   /* Pętla do/while(FALSE) ułatwia realizację procedur czyszczących w    */
   /* przypadku błędu. Funkcja close() dla poszczególnych deskryptorów    */
   /* gniazd jest uruchamiana jednokrotnie na samym końcu programu.       */
   /***********************************************************************/
   do
   {
      /********************************************************************/
      /* Funkcja socket() zwraca deskryptor gniazda stanowiącego punkt    */
      /* końcowy. Instrukcja ta określa także, że dla tego gniazda        */
      /* użyta zostanie rodzina adresów INET (Internet Protocol)          */
```

```
   /* z protokołem transportowym UDP (SOCK_DGRAM).                    */
   /********************************************************************/
   sd = socket(AF_INET, SOCK_DGRAM, 0);
   if (sd < 0)
 {
      perror("Niepowodzenie socket()");
      break;
   }

   /********************************************************************/
   /* Po utworzeniu deskryptora gniazda funkcja bind() pobiera         */
   /* unikalną nazwę gniazda. W tym przykładzie użytkownik ustawia     */
   /* wartość s_addr na zero, co oznacza, że port UDP 3555 będzie      */
   /* powiązany z wszystkimi adresami IP w systemie.                   */
   /********************************************************************/
   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("Niepowodzenie bind()");
      break;
   }

   /********************************************************************/
   /* Do odebrania tych danych serwer używa funkcji recvfrom().        */
   /* Funkcja recvfrom() czeka na dane przez czas nieokreślony.        */
   /********************************************************************/
   rc = recvfrom(sd, buffer, sizeof(buffer), 0,
                 (struct sockaddr *)&clientaddr,
                 &clientaddrlen);
   if (rc < 0)
   {
      perror("Niepowodzenie recvfrom()");
      break;
   }

   printf("serwer wysłał: <%s>\n", buffer);
   printf("z portu %d, adresu %s\n",
          ntohs(clientaddr.sin_port),
          inet_ntoa(clientaddr.sin_addr));

   /********************************************************************/
   /* Odesłanie danych do klienta               */
   /********************************************************************/
   rc = sendto(sd, buffer, sizeof(buffer), 0,
               (struct sockaddr *)&clientaddr,
               sizeof(clientaddr));
   if (rc < 0)
   {
      perror("Niepowodzenie sendto()");
      break;
   }

   /********************************************************************/
   /* Zakończenie programu                                             */
   /********************************************************************/

} while (FALSE);

/***********************************************************************/
/* Zamknięcie wszystkich otwartych deskryptorów gniazd                 */
```

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

Przykład: bezpołączeniowy program klienta

Poniższy przykład ilustruje użycie protokołu UDP do łączenia bezpołączeniowego programu klienta używającego gniazd z serwerem. Informacje dotyczące wykorzystania przykładowych kodów zawiera sekcja Informacje dotyczące kodu.

```
/***********************************************************************/
/* Jest to przykładowy kod programu klienta bezpołączeniowego.          */
/***********************************************************************/

/***********************************************************************/
/* Wymagane przez program pliki nagłówkowe.                              */
/***********************************************************************/
#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>

/***********************************************************************/
/* Stałe używane przez ten program                                      */
/***********************************************************************/
#define SERVER_PORT     3555
#define BUFFER_LENGTH    100
#define FALSE              0
#define SERVER_NAME     "NazwaSerweraHosta"

/* Przekaż 1 parametr, który albo jest     */
/* adresem, albo nazwą hosta serwera lub   */
/* ustaw nazwę serwera zmienną #define     */
/* SERVER_NAME.                            */
void main(int argc, char *argv[])
{
   /********************************************************************/
   /* Definicje zmiennych i struktur.                                  */
   /********************************************************************/
   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);

   /********************************************************************/
   /* Pętla do/while(FALSE) ułatwia realizację procedur czyszczących w */
   /* przypadku błędu. Funkcja close() dla deskryptora gniazda jest    */
   /* uruchamiana jednokrotnie na samym końcu programu.                */
   /********************************************************************/
   do
   {
      /*****************************************************************/
      /* Funkcja socket() zwraca deskryptor gniazda stanowiącego punkt */
      /* końcowy. Instrukcja ta określa także, że dla tego gniazda     */
      /* użyta zostanie rodzina adresów INET (Internet Protocol)       */
      /* z protokołem transportowym TCP (SOCK_STREAM).                 */
      /*****************************************************************/
      sd = socket(AF_INET, SOCK_DGRAM, 0);
      if (sd < 0)
    {
         perror("Niepowodzenie socket()");
         break;
```

```
}

/********************************************************************/
/* Jeśli został przekazany argument, należy go użyć jako nazwy      */
/* serwera, w przeciwnym razie należy użyć zmiennej określonej      */
/* w makrze #define znajdującym się na początku programu.           */
/********************************************************************/
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)
{
   /*****************************************************************/
   /* Łańcuch określający serwer, przekazany do funkcji inet_addr() */
   /* nie jest adresem IP składającym się z liczb dziesiętnych      */
   /* oddzielonych kropkami, zatem musi być to nazwa hosta serwera. */
   /* Do pobrania adresu IP serwera zostanie użyta funkcja          */
   /* gethostbyname().                                              */
   /*****************************************************************/
   hostp = gethostbyname(server);
   if (hostp == (struct hostent *)NULL)
   {
      printf("Hosta nie znaleziono --> ");
      printf("h_errno = %d\n", h_errno);
      break;
   }

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

/********************************************************************/
/* Zainicjowanie bloku danych, który zostanie wysłany do serwera    */
/********************************************************************/
memset(buffer, 0, sizeof(buffer));
strcpy(buffer, "A CLIENT REQUEST");

/********************************************************************/
/* Do wysłania danych do serwera zostanie użyta funkcja sendto()    */
/********************************************************************/
rc = sendto(sd, buffer, sizeof(buffer), 0,
            (struct sockaddr *)&serveraddr,
            sizeof(serveraddr));
if (rc < 0)
{
   perror("Niepowodzenie sendto()");
   break;
}

/********************************************************************/
/* Do odebrania tych danych z serwera zostanie użyta funkcja        */
/* recvfrom()                                                       */
/********************************************************************/
rc = recvfrom(sd, buffer, sizeof(buffer), 0,
              (struct sockaddr *)&serveraddr,
              & serveraddrlen);
if (rc < 0)
{
   perror("Niepowodzenie recvfrom()");
   break;
```

```
        }

        printf("klient odebrał: <%s>\n", buffer);
        printf(" z portu %d, adresu %s\n",
               ntohs(serveraddr.sin_port),
               inet_ntoa(serveraddr.sin_addr));

        /*****************************************************************/
        /* Zakończenie programu                                          */
        /*****************************************************************/

     } while (FALSE);

     /*******************************************************************/
     /* Zamknięcie wszystkich otwartych deskryptorów gniazd             */
     /*******************************************************************/
     if (sd != -1)
        close(sd);
}
```

Projektowanie aplikacji używających rodzin adresów

Poniższe sekcje zawierają przykłady programów, które ilustrują każdą rodzinę adresów gniazd:

- Używanie rodziny adresów AF_INET
- Używanie rodziny adresów AF_INET6
- Używanie rodziny adresów AF_UNIX
- Używanie rodziny adresów AF_TELEPHONY
- Używanie rodziny adresów AF_UNIX_CCSID

Używanie rodziny adresów AF_INET

Gniazda rodziny adresów AF_INET mogą być zorientowane na połączenie (typu SOCK_STREAM), jak i bezpołączeniowe (typu SOCK_DGRAM). Zorientowane na połączenie gniazda AF_INET używają TCP jako protokołu transportowego. Bezpołączeniowe gniazda AF_INET jako protokołu transportowego używają UDP. Po utworzeniu gniazda domeny AF_INET w programie używającym gniazd podaje się AF_INET jako rodzinę adresów. Gniazda AF_INET mogą być również typu SOCK_RAW. W takim przypadku aplikacja łączy się bezpośrednio z warstwą IP i nie używa transportu TCP ani UDP.

Sekcja Wymagania wstępne do programowania z użyciem gniazd zawiera szczegółowe informacje dotyczące konfigurowania środowiska do używania rodziny adresów AF_INET.

Przykładowe programy korzystające z rodziny adresów AF_INET znajdują się w sekcjach Przykład: program serwera zorientowany na połączenie i Przykład: program klienta zorientowany na połączenie.

Używanie rodziny adresów AF_INET6

Gniazda AF_INET6 zapewniają obsługę 128-bitowych (16-bajtowych) struktur adresów protokołu IP (Internet Protocol) w wersji 6 (IPv6). Programiści mogą pisać aplikacje używające rodziny adresów AF_INET6, które będą akceptowały połączenia z klientów obsługujących zarówno protokół IPv4, jak i IPv6 lub też od klientów obsługujących tylko protokół IPv6.

Podobnie jak gniazda rodziny adresów AF_INET, gniazda AF_INET6 mogą być zorientowane na połączenie (typu SOCK_STREAM), jak i bezpołączeniowe (typu SOCK_DGRAM). Zorientowane na połączenie gniazda AF_INET6 jako protokołu transportowego używają TCP. Bezpołączeniowe gniazda AF_INET6 jako protokołu transportowego używają UDP. Po utworzeniu gniazda domeny AF_INET6 w programie używającym gniazd podaje się AF_INET6 jako rodzinę adresów. Gniazda AF_INET6 mogą być również typu SOCK_RAW. W takim przypadku aplikacja łączy się bezpośrednio z warstwą IP i nie używa transportu TCP ani UDP. Sekcja Wymagania wstępne do programowania z użyciem gniazd zawiera szczegółowe informacje dotyczące konfigurowania środowiska do używania rodziny adresów AF_INET6.

Zgodność aplikacji IPv6 z aplikacjami IPv4

Aplikacje gniazd napisane dla rodziny adresów AF_INET6, używające protokołu IP (Internet Protocol) wersja 6 (IPv6), współpracują z aplikacjami dla protokołu IP (Internet Protocol) wersja 4 (IPv4), czyli używającymi rodziny adresów AF_INET. Dzięki temu programiści piszący aplikacje gniazd mogą korzystać z formatu adresu IPv4 odwzorowanego na adres IPv6. Format ten polega na tym, że adresowi IPv4 węzła IPv4 odpowiada adres IPv6. Adres IPv4 jest zapisywany w najmłodszych 32 bitach adresu IPv6, a najstarsze 96 bitów stanowi ustalony przedrostek 0:0:0:0:0:FFFF. Przykładowy odwzorowany adres IPv4 wygląda następująco:

```
::FFFF:192.1.1.1
```

Adresy te są generowane przez funkcję **getaddrinfo()** automatycznie wtedy, gdy podany host ma wyłącznie adresy IPv4.

Do otwierania połączeń TCP z węzłami IPv4 można używać aplikacji napisanych z użyciem gniazd AF_INET6. W tym celu można zakodować adres IPv4 miejsca docelowego jako adres IPv4 odwzorowany na adres IPv6 i przekazać go w strukturze sockaddr_in6 do wywołania funkcji **connect()** lub **sendto()** . Gdy aplikacje używają gniazd AF_INET6 do odbierania połączeń TCP z węzłów IPv4 lub odbierają pakiety UDP z węzłów IPv4, system zwraca adres węzła do aplikacji w wywołaniach funkcji **accept()**, **recvfrom()** lub **getpeername()** przy użyciu struktury sockaddr_in6 zakodowanej w opisany wyżej sposób.

Mimo iż funkcja **bind()** umożliwia aplikacjom wybranie źródłowego adresu IP pakietów UDP i połączeń TCP, aplikacje często żądają, aby system wybrał adres źródłowy. W tym celu używają struktury in6addr_any w podobny sposób, jak makra INADDR_ANY w protokole IPv4. Dodatkową zaletą takiego wiązania jest możliwość komunikacji pomiędzy gniazdem AF_INET6 a węzłami IPv4 i IPv6. Na przykład, aplikacja wywołująca funkcję **accept()** na nasłuchującym gnieździe powiązanym z in6addr_any będzie akceptowała połączenia zarówno z węzłów IPv4, jak i IPv6. Zachowanie takie można modyfikować za pomocą opcji gniazd IPV6_V6ONLY na poziomie IPPROTO_IPV6. Jest prawdopodobne, że niektóre aplikacje będą musiały mieć informację, z jakim typem węzła współpracują. Dla takich aplikacji dostępne jest makro IN6_IS_ADDR_V4MAPPED() zdefiniowane w pliku <netinet/in.h>.

Bardziej szczegółowe porównanie protokołów IPv4 i IPv6 zawiera temat Porównanie protokołów IPv4 i IPv6 w Centrum informacyjnym. Zawiera on porównanie opcji obu protokołów.

Przykładowe programy i opisy sytuacji, gdzie gniazdo AF_INET6 jednocześnie komunikuje się z węzłami IPv4 i IPv6 zawiera rozdział Tworzenie aplikacji akceptującej klientów IPv4 i IPv6.

Ograniczenia protokołu IPv6

W wersji V5R2 systemu OS/400 obsługa protokołu IPv6 jest do pewnego stopnia ograniczona. Poniższa tabela zawiera listę tych ograniczeń i wynikających z nich kwestii związanych z pisaniem aplikacji gniazd.

Tabela 12. Ograniczenia protokołu IPv6 i ich skutki

Ograniczenie	Skutek
Protokół IPv6 nie obsługuje fragmentacji.	Gniazda AF_INET6 (SOCK_DGRAM) nie powinny wysyłać datagramów większych niż jednostka MTU interfejsu minus wielkość nagłówków.
Funkcja anycast nie jest w protokole IPv6 obsługiwana.	Nie można łączyć się z adresami anycast ani wysyłać do nich pakietów.
Rozsyłanie grupowe nie jest w protokole IPv6 obsługiwane.	Nie można wysyłać ani odbierać datagramów rozsyłania grupowego.
Tabela hostów systemu iSeries nie obsługuje adresów IPv6.	Funkcje API **getaddrinfo()** i **getnameinfo()** nie będą w stanie znaleźć adresów w tabeli hostów. Możliwe jest jedynie wyszukiwanie adresów przy użyciu serwera DNS.
Funkcje API **gethostbyname()** i **gethostbyaddr()** obsługują jedynie rozstrzyganie adresów protokołu IPv4.	Jeśli wymagane jest rozstrzyganie adresów protokołu IPv6, należy użyć funkcji API **getaddrinfo()** i **getnameinfo()**.

Używanie rodziny adresów AF_UNIX

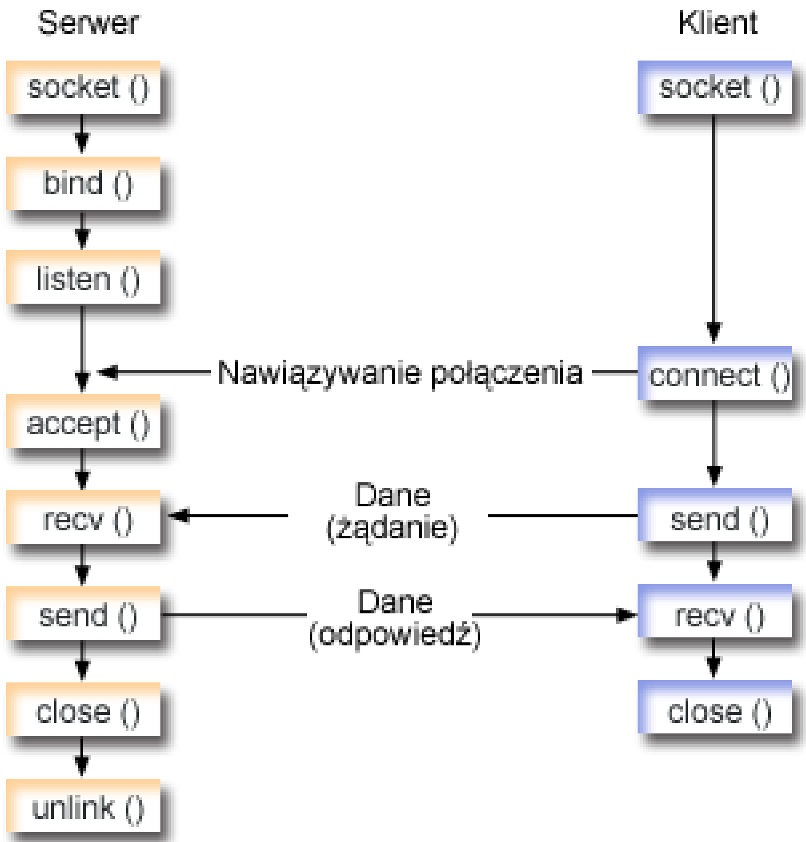
Gniazda rodziny adresów AF_UNIX (gniazda używające rodzin adresów AF_UNIX lub AF_UNIX_CCSID) mogą być zorientowane na połączenie (typu SOCK_STREAM) lub bezpołączeniowe (typu SOCK_DGRAM). Ponieważ żadna zewnętrzna funkcja komunikacyjna nie łączy dwóch procesów, obydwa typy są tak samo niezawodne.

Gniazda datagramowe domeny UNIX działają inaczej niż gniazda datagramowe UDP. W przypadku gniazda datagramowego UDP program typu klient nie musi wywoływać funkcji **bind()**, gdyż system automatycznie przypisze numer nie używanego portu. Serwer może następnie wysłać datagram z powrotem na ten numer portu. Jednak w przypadku gniazd datagramowych domeny UNIX system nie przypisze automatycznie nazwy ścieżki dla klienta. Dlatego wszystkie programy typu klient korzystające z datagramów domeny UNIX, muszą wywołać funkcję **bind()**. Dokładna nazwa ścieżki podana w funkcji **bind()** klienta jest wysyłana do serwera. Dlatego, jeśli klient określi względną nazwę ścieżki (to jest nazwę ścieżki, która nie zaczyna się od ukośnika - /), serwer nie będzie mógł odesłać datagramu klienta, chyba że jest on wykonywany w tym samym katalogu bieżącym.

Przykładowa nazwa ścieżki, której aplikacja może użyć dla tej rodziny adresów, to /tmp/mójserwer lub serwery/tamtenserwer. Nazwa serwery/tamtenserwer to nazwa ścieżki, która nie jest pełna (nie zaczyna się od znaku /). Oznacza to, że położenie pozycji w hierarchii systemu plików powinno zostać określone w odniesieniu do bieżącego katalogu roboczego.

Uwaga: Nazwy ścieżek w systemie plików obsługują narodowe wersje językowe.

Poniższy rysunek ilustruje relację klient/serwer rodziny adresów AF_UNIX. Sekcja Wymagania wstępne do programowania z użyciem gniazd zawiera szczegółowe informacje dotyczące konfigurowania środowiska do używania rodziny adresów AF_UNIX.



Przebieg zdarzeń w gnieździe: aplikacja serwera używająca rodziny adresów AF_UNIX
W sekcji Przykład: aplikacja serwera używająca rodziny adresów AF_UNIX użyto wywołań następujących funkcji:

1. Funkcja **socket()** zwraca deskryptor gniazda odpowiadający punktowi końcowemu. Instrukcja ta informuje również, że dla tego gniazda zostanie użyta rodzina adresówUNIX z transportem strumieniowym (SOCK_STREAM).Funkcja ta zwraca deskryptor gniazda odpowiadający punktowi końcowemu. Do zainicjowania gniazda UNIX można także użyć funkcji **socketpair()**.

   AF_UNIX i AF_UNIX_CCSID to jedyne rodziny adresów, które obsługują funkcję **socketpair()**. Funkcja **socketpair()** zwraca dwa nienazwane i podłączone deskryptory gniazd.
2. Po utworzeniu deskryptora gniazda funkcja **bind()** pobiera unikalną nazwę gniazda.

   Przestrzeń nazw dla gniazd domeny UNIX składa się z nazw ścieżek. Kiedy program używający gniazd wywołuje funkcję **bind()**, tworzona jest pozycja katalogu systemu plików. Jeśli nazwa ścieżki już istnieje, **bind()** nie powiedzie się. Dlatego program używający gniazda domeny UNIX powinien zawsze wywoływać funkcję **unlink()**, aby po zakończeniu działania usunąć tę pozycję katalogu.
3. Funkcja **listen()** umożliwia serwerowi przyjęcie połączenia przychodzącego od klienta. W tym przykładzie kolejka (backlog) ma wartość 10. Oznacza to, że system umieści w kolejce pierwsze 10 przychodzących żądań połączenia, a następne odrzuci.
4. Funkcja **recv()** odbiera dane z aplikacji klienta. W tym przykładzie wiadomo, że klient wyśle 250 bajtów danych. Na tej podstawie można użyć opcji gniazda SO_RCVLOWAT i określić, że funkcja **recv()** ma pozostać w uśpieniu do momentu nadejścia całych 250 bajtów danych.
5. Funkcja **send()** odsyła dane do klienta.
6. Funkcja **close()** zamyka wszystkie otwarte deskryptory gniazd.
7. Funkcja **unlink()** usuwa nazwę ścieżkiUNIX z systemu plików.

Przebieg zdarzeń w gnieździe: aplikacja klienta używająca rodziny adresów AF_UNIX

W sekcji Przykład: aplikacja klienta używająca rodziny adresów AF_UNIX użyto wywołań następujących funkcji:

1. Funkcja **socket()** zwraca deskryptor gniazda odpowiadający punktowi końcowemu. Instrukcja ta informuje również, że dla tego gniazda zostanie użyta rodzina adresówUNIX z transportem strumieniowym (SOCK_STREAM).Funkcja ta zwraca deskryptor gniazda odpowiadający punktowi końcowemu. Do zainicjowania gniazda UNIX można także użyć funkcji **socketpair()**.

   AF_UNIX i AF_UNIX_CCSID to jedyne rodziny adresów, które obsługują funkcję **socketpair()**. Funkcja **socketpair()** zwraca dwa nienazwane i podłączone deskryptory gniazd.
2. Po odebraniu deskryptora gniazda należy użyć funkcji **connect()** do nawiązania połączenia z serwerem.
3. Funkcja **send()** wysyła 250 bajtów danych określonych w aplikacji serwera przez opcję gniazda SO_RCVLOWAT.
4. Funkcja **recv()** wykonuje wielokrotnie pętlę do momentu, gdy zostanie odebrane całe 250 bajtów.
5. Funkcja **close()** zamyka wszystkie otwarte deskryptory gniazd.

Przykład: aplikacja serwera używająca rodziny adresów AF_UNIX: Poniższy przykład przedstawia serwer dla rodziny adresów AF_UNIX. Rodzina adresów AF_UNIX używa wielu tych samych wywołań funkcji gniazd co inne rodziny adresów z tym, że do identyfikacji aplikacji serwera używa struktury nazwy ścieżki. Poniższe programy przykładowe używają rodziny adresów AF_UNIX. Informacje dotyczące wykorzystania przykładowych kodów zawiera sekcja Informacje dotyczące kodu.

```
/**************************************************************************/
/* Ten przykładowy kod jest aplikacją serwera, używającą rodziny adresów  */
/* AF_UNIX                                                                */
/**************************************************************************/

/**************************************************************************/
/* Wymagane przez program pliki nagłówkowe.                               */
/**************************************************************************/
#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/un.h>

/**************************************************************************/
/* Stałe używane przez ten program                                        */
```

```
/**************************************************************************/
#define SERVER_PATH     "/tmp/server"
#define BUFFER_LENGTH    250
#define FALSE              0

void main()
{
   /***********************************************************************/
   /* Definicje zmiennych i struktur.                                     */
   /***********************************************************************/
   int    sd=-1, sd2=-1;
   int    rc, length;
   char   buffer[BUFFER_LENGTH];
   struct sockaddr_un serveraddr;

   /***********************************************************************/
   /* Pętla do/while(FALSE) ułatwia realizację procedur czyszczących w    */
   /* przypadku błędu. Funkcja close() dla poszczególnych deskryptorów    */
   /* gniazd jest uruchamiana jednokrotnie na samym końcu programu.       */
   /***********************************************************************/
   do
   {
      /********************************************************************/
      /* Funkcja socket() zwraca deskryptor gniazda stanowiącego punkt    */
      /* końcowy. Instrukcja ta określa także, że dla tego gniazda        */
      /* użyta zostanie rodzina adresów UNIX_CCSID ze strumieniowym       */
      /* protokołem transportowym (SOCK_DGRAM).                           */
      /********************************************************************/
      sd = socket(AF_UNIX, SOCK_STREAM, 0);
      if (sd < 0)
    {
         perror("Niepowodzenie socket()");
         break;
      }

      /********************************************************************/
      /* Po utworzeniu deskryptora gniazda funkcja bind() pobiera         */
      /* unikalną nazwę gniazda.                                          */
      /********************************************************************/
      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("Niepowodzenie bind()");
         break;
      }

      /********************************************************************/
      /* Funkcja listen() umożliwia serwerowi przyjęcie połączeń          */
      /* przychodzących od klienta. W tym przykładzie kolejka (backlog)   */
      /* ma wartość 10. Oznacza to, że system umieści w kolejce pierwsze  */
      /* 10 przychodzących żądań połączenia, a następne będzie je         */
      /* odrzucał.                                                        */
      /********************************************************************/
      rc = listen(sd, 10);
      if (rc < 0)
      {
         perror("Niepowodzenie listen()");
         break;
      }

      printf("Gotowy do obsługi klienta (connect()).\n");

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

```
      /* Serwer używa funkcji accept() do zaakceptowania połączenia      */
      /* przychodzącego. Wywołanie funkcji accept() zostanie zablokowane  */
      /* na nieokreślony czas oczekiwania na połączenie przychodzące.     */
      /********************************************************************/
      sd2 = accept(sd, NULL, NULL);
      if (sd2 < 0)
      {
         perror("Niepowodzenie accept()");
         break;
      }

      /********************************************************************/
      /* W tym przykładzie wiadomo, że klient wyśle 250 bajtów danych.    */
      /* Dzięki temu można użyć opcji gniazda SO_RCVLOWAT i określić, że  */
      /* funkcja recv() ma nie wychodzić z uśpienia, dopóki nie zostanie  */
      /* odebrane wszystkie 250 bajtów danych.                            */
      /********************************************************************/
      length = BUFFER_LENGTH;
      rc = setsockopt(sd2, SOL_SOCKET, SO_RCVLOWAT,
                                          (char *)&length, sizeof(length));
      if (rc < 0)
      {
      }

      printf("Otrzymano dane, bajtów: %d\n", rc);
      if (rc == 0 ||
          rc < sizeof(buffer))
      {
         printf("Klient zamknął połączenie przed wysłaniem\n");
         printf("wszystkich danych\n");
         break;
      }

      /********************************************************************/
      /* Odesłanie danych do klienta                */
      /********************************************************************/
      rc = send(sd2, buffer, sizeof(buffer), 0);
      if (rc < 0)
      {
         perror("Niepowodzenie send()");
         break;
      }

      /********************************************************************/
      /* Zakończenie programu                                             */
      /********************************************************************/

   } while (FALSE);

   /***********************************************************************/
   /* Zamknięcie wszystkich otwartych deskryptorów gniazd                 */
   /***********************************************************************/
   if (sd != -1)
      close(sd);

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

    /***********************************************************************/
   /* Usunięcie ścieżki UNIX z systemu plików.                            */
   /***********************************************************************/
   unlink(SERVER_PATH);
}
```

Przykład: aplikacja klienta używająca rodziny adresów AF_UNIX: Poniższy przykład przedstawia aplikację klienta dla rodziny adresów AF_UNIX. Rodzina adresów AF_UNIX używa wielu tych samych wywołań funkcji gniazd co inne rodziny adresów z tym, że do identyfikacji aplikacji serwera używa struktury nazwy ścieżki. Poniższy

program przykładowy używa rodziny adresów AF_UNIX do nawiązania połączenia klienta z serwerem. Informacje dotyczące wykorzystania przykładowych kodów zawiera sekcja Informacje dotyczące kodu.

```
/**************************************************************************/
/* Ten przykładowy kod jest aplikacją klienta, używającą rodziny adresów  */
/* AF_UNIX                                                                */
/**************************************************************************/
/**************************************************************************/
/* Wymagane przez program pliki nagłówkowe.                               */
/**************************************************************************/
#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/un.h>

/**************************************************************************/
/* Stałe używane przez ten program                                        */
/**************************************************************************/
#define SERVER_PATH     "/tmp/server"
#define BUFFER_LENGTH    250
#define FALSE              0

/* Przekaż 1 parametr, który albo jest     */
/* nazwą ścieżki serwera w postaci ciągu   */
/* znaków UNICODE, albo ustawioną w makrze */
/* #define SERVER_PATH ścieżką serwera,    */
/* która jest łańcuchem CCSID 500.         */
void main(int argc, char *argv[])
{
   /***********************************************************************/
   /* Definicje zmiennych i struktur.                                     */
   /***********************************************************************/
   int    sd=-1, rc, bytesReceived;
   char   buffer[BUFFER_LENGTH];
   struct sockaddr_un serveraddr;

   /***********************************************************************/
   /* Pętla do/while(FALSE) ułatwia realizację procedur czyszczących w    */
   /* przypadku błędu. Funkcja close() dla deskryptora gniazda jest       */
   /* uruchamiana jednokrotnie na samym końcu programu.                   */
   /***********************************************************************/
   do
   {
      /********************************************************************/
      /* Funkcja socket() zwraca deskryptor gniazda stanowiącego punkt    */
      /* końcowy. Instrukcja ta określa także, że dla dla tego gniazda    */
      /* użyta zostanie rodzina adresów UNIX_CCSID ze strumieniowym       */
      /* protokołem transportowym (SOCK_DGRAM).                           */
      /********************************************************************/
      sd = socket(AF_UNIX_CCSID, SOCK_STREAM, 0);
      if (sd < 0)
    {
         perror("Niepowodzenie socket()");
         break;
      }

      /********************************************************************/
      /* Jeśli został przekazany argument, należy go użyć jako nazwy      */
      /* serwera, w przeciwnym razie należy użyć zmiennej określonej      */
      /* w makrze #define znajdującym się na początku programu.           */
      /********************************************************************/
      memset(&serveraddr, 0, sizeof(serveraddr));
      serveraddr.sun_family = AF_UNIX;
      if (argc > 1)
         strcpy(serveraddr.sun_path, argv[1]);
      else
```

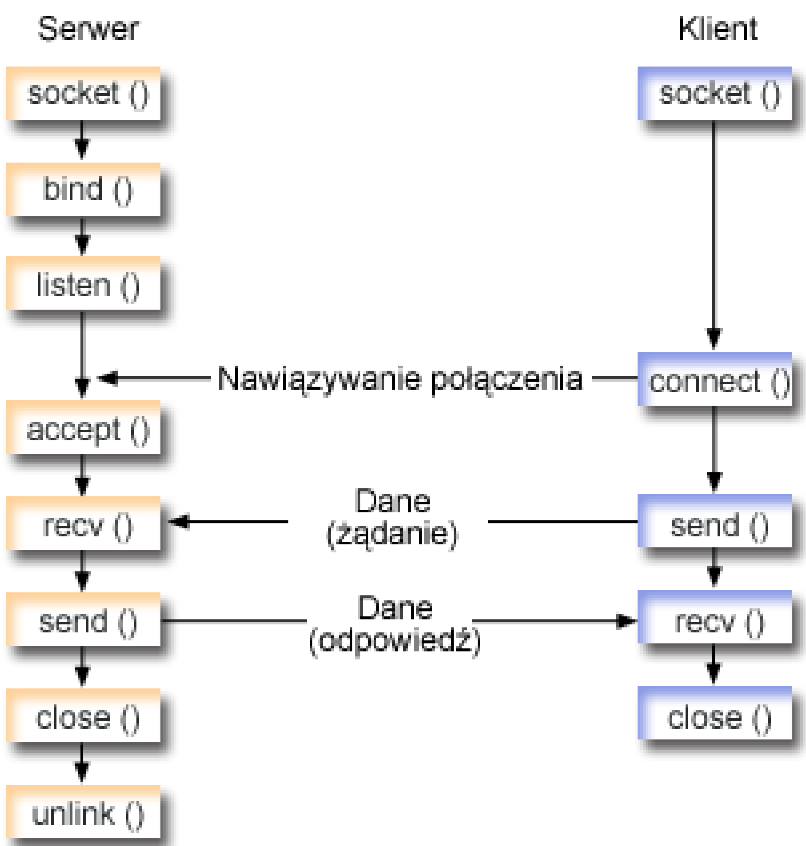
```
         strcpy(serveraddr.sun_path, SERVER_PATH);

      /********************************************************************/
      /* Aby nawiązać połączenie z serwerem, zostanie użyta funkcja       */
      /* connect().                                                       */
      /********************************************************************/
      rc = connect(sd, (struct sockaddr *)&serveraddr, SUN_LEN(&serveraddr));
      if (rc < 0)
      {
         perror("Niepowodzenie connect()");
         break;
      }

      /********************************************************************/
      /* Wysłanie 250 bajtów znaków 'a' do serwera                        */
      /********************************************************************/
      memset(buffer, 'a', sizeof(buffer));
      rc = send(sd, buffer, sizeof(buffer), 0);
      if (rc < 0)
      {
         perror("Niepowodzenie send()");
         break;
      }

      /********************************************************************/
      /* W tym przykładzie wiadomo, że serwer odpowie wysłaniem tych      */
      /* samych 250 bajtów, które wysłaliśmy. Ponieważ wiadomo, że        */
      /* zostanie odesłanych 250 bajtów, można użyć opcji gniazda         */
      /* SO_RCVLOWAT, uruchomić pojedynczą funkcję recv() i pobrać        */
      /* wszystkie dane.                                                  */
      /*                                                                  */
      /* Użycie opcji SO_RCVLOWAT zostało już pokazane w przykładzie      */
      /* serwera, dlatego tutaj użyjemy innej metody. Ponieważ te 250     */
      /* bajtów danych może być przysłanych w oddzielnych pakietach,      */
      /* będziemy wielokrotnie uruchamiali funkcję recv(), dopóki         */
      /* nie odbierzemy wszystkich 250 bajtów.                            */
      /********************************************************************/
      bytesReceived = 0;
      while (bytesReceived < BUFFER_LENGTH)
      {
         rc = recv(sd, & buffer[bytesReceived],
                   BUFFER_LENGTH - bytesReceived, 0);
         if (rc < 0)
         {
            perror("Niepowodzenie recv()");
            break;
         }
         else if (rc == 0)
         {
            printf("Serwer zamknął połączenie\n");
            break;
         }

         /*****************************************************************/
         /* Zwiększenie liczby otrzymanych dotychczas bajtów              */
         /*****************************************************************/
         bytesReceived += rc;
      }

   } while (FALSE);

   /***********************************************************************/
   /* Zamknięcie wszystkich otwartych deskryptorów gniazd                 */
   /***********************************************************************/
   if (sd != -1)
      close(sd);
}
```

Używanie rodziny adresów AF_UNIX_CCSID

Gniazdo rodziny adresów AF_UNIX_CCSID ma takie same specyfikacje, co gniazda rodziny adresów AF_UNIX. Mogą być one używane do komunikacji w trybie zorientowanym na połączenie i bezpołączeniowym, a także do komunikacji w tym samym systemie. Szczegóły zawiera sekcja Korzystanie z rodziny adresów AF_UNIX.

Aby rozpocząć pracę z aplikacją używającą gniazd AF_UNIX_CCSID, należy zapoznać się ze strukturą **Qlg_Path_Name_T**, w celu określenia formatu wyjściowego. Szczegóły zawiera sekcja Struktury nazw ścieżek w Skorowidzu funkcji API, w Centrum informacyjnym.

Podczas pracy ze strukturą adresu wyjściowego, na przykład zwróconą przez funkcje **accept()**, **getsockname()**, **getpeername()**, **recvfrom()** lub **recvmsg()**, aplikacja musi sprawdzić strukturę adresu gniazda (sockaddr_unc), aby określić jego format. Wyjściowy format nazwy ścieżki określają pola sunc_format i sunc_qlg. Jednak gniazda nie zawsze używają tych samych wartości na wyjściu, co aplikacje na wejściu.



Przebieg zdarzeń w gnieździe: aplikacja serwera używająca rodziny adresów AF_UNIX_CCSID

W sekcji Przykład: aplikacja serwera używająca rodziny adresów AF_UNIX_CCSID użyto wywołań następujących funkcji:

1. Funkcja **socket()** zwraca deskryptor gniazda odpowiadający punktowi końcowemu. Instrukcja ta informuje również, że dla tego gniazda zostanie użyta rodzina adresów UNIX_CCSID z transportem strumieniowym (SOCK_STREAM). Do zainicjowania gniazda UNIX można także użyć funkcji **socketpair()**.

   AF_UNIX i AF_UNIX_CCSID to jedyne rodziny adresów, które obsługują funkcję **socketpair()**. Funkcja **socketpair()** zwraca dwa nienazwane i podłączone deskryptory gniazd.
2. Po utworzeniu deskryptora gniazda funkcja **bind()** pobiera unikalną nazwę gniazda.

Przestrzeń nazw dla gniazd domeny UNIX składa się z nazw ścieżek. Kiedy program używający gniazd wywołuje funkcję **bind()**, tworzona jest pozycja katalogu systemu plików. Jeśli nazwa ścieżki już istnieje, **bind()** nie powiedzie się. Dlatego program używający gniazda domeny UNIX powinien zawsze wywoływać funkcję **unlink()**, aby po zakończeniu działania usunąć tę pozycję katalogu.

3. Funkcja **listen()** umożliwia serwerowi przyjęcie połączenia przychodzącego od klienta. W tym przykładzie kolejka (backlog) ma wartość 10. Oznacza to, że system umieści w kolejce pierwsze 10 przychodzących żądań połączenia, a następne odrzuci.
4. Serwer używa funkcji **accept()** do zaakceptowania żądania połączenia przychodzącego. Wywołanie funkcji **accept()** zostanie zablokowane na nieokreślony czas oczekiwania na połączenie przychodzące.
5. Funkcja **recv()** odbiera dane z aplikacji klienta. W tym przykładzie wiadomo, że klient wyśle 250 bajtów danych. Na tej podstawie można użyć opcji gniazda SO_RCVLOWAT i określić, że funkcja **recv()** ma pozostać w uśpieniu do momentu nadejścia całych 250 bajtów danych.
6. Funkcja **send()** odsyła dane do klienta.
7. Funkcja **close()** zamyka wszystkie otwarte deskryptory gniazd.
8. Funkcja **unlink()** usuwa nazwę ścieżkiUNIX z systemu plików.

Przebieg zdarzeń w gnieździe: aplikacja klienta używająca rodziny adresów AF_UNIX_CCSID
W sekcji Przykład: aplikacja klienta używająca rodziny adresów AF_UNIX_CCSID użyto wywołań następujących funkcji:

1. Funkcja **socket()** zwraca deskryptor gniazda odpowiadający punktowi końcowemu. Instrukcja ta informuje również, że dla tego gniazda zostanie użyta rodzina adresówUNIX z transportem strumieniowym (SOCK_STREAM).Funkcja ta zwraca deskryptor gniazda odpowiadający punktowi końcowemu. Do zainicjowania gniazda UNIX można także użyć funkcji **socketpair()**.

   AF_UNIX i AF_UNIX_CCSID to jedyne rodziny adresów, które obsługują funkcję **socketpair()**. Funkcja **socketpair()** zwraca dwa nienazwane i podłączone deskryptory gniazd.
2. Po odebraniu deskryptora gniazda należy użyć funkcji **connect()** do nawiązania połączenia z serwerem.
3. Funkcja **send()** wysyła 250 bajtów danych określonych w aplikacji serwera przez opcję gniazda SO_RCVLOWAT.
4. Funkcja **recv()** wykonuje wielokrotnie pętlę do momentu, gdy zostanie odebrane całe 250 bajtów.
5. Funkcja **close()** zamyka wszystkie otwarte deskryptory gniazd.

Przykład: aplikacja serwera używająca rodziny adresów AF_UNIX_CCSID: Poniższe programy przykładowe używają rodziny adresów AF_UNIX_CCSID. Informacje dotyczące wykorzystania przykładowych kodów zawiera sekcja Informacje dotyczące kodu.

```
/**************************************************************************/
/* Ten przykładowy kod jest aplikacją serwera, używającą rodziny adresów  */
/* AF_UNIX_CCSID.                                                         */
/**************************************************************************/

/**************************************************************************/
/* Wymagane przez program pliki nagłówkowe.                               */
/**************************************************************************/
#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/unc.h>

/**************************************************************************/
/* Stałe używane przez ten program                                        */
/**************************************************************************/
#define SERVER_PATH     "/tmp/server"
#define BUFFER_LENGTH    250
#define FALSE              0

void main()
{
   /***********************************************************************/
```

```
/* Definicje zmiennych i struktur.                                    */
/**********************************************************************/
int    sd=-1, sd2=-1;
int    rc, length;
char   buffer[BUFFER_LENGTH];
struct sockaddr_unc serveraddr;

/**********************************************************************/
/* Pętla do/while(FALSE) ułatwia realizację procedur czyszczących w    */
/* przypadku błędu. Funkcja close() dla poszczególnych deskryptorów    */
/* gniazd jest uruchamiana jednokrotnie na samym końcu programu.       */
/**********************************************************************/
do
{
   /*******************************************************************/
   /* Funkcja socket() zwraca deskryptor gniazda stanowiącego punkt    */
   /* końcowy. Instrukcja ta określa także, że dla dla tego gniazda    */
   /* użyta zostanie rodzina adresów UNIX_CCSID ze strumieniowym       */
   /* protokołem transportowym (SOCK_DGRAM).                           */
   /*******************************************************************/
   sd = socket(AF_UNIX_CCSID, SOCK_STREAM, 0);
   if (sd < 0)
  {
      perror("Niepowodzenie socket()");
      break;
   }

   /*******************************************************************/
   /* Po utworzeniu deskryptora gniazda funkcja bind() pobiera         */
   /* unikalną nazwę gniazda.                                          */
   /*******************************************************************/
   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("Niepowodzenie bind()");
      break;
   }

   /*******************************************************************/
   /* Funkcja listen() umożliwia serwerowi przyjęcie połączeń          */
   /* przychodzących od klienta. W tym przykładzie kolejka (backlog)   */
   /* ma wartość 10. Oznacza to, że system umieści w kolejce pierwsze  */
   /* 10 przychodzących żądań połączenia, a następne będzie je         */
   /* odrzucał.                                                        */
   /*******************************************************************/
   rc = listen(sd, 10);
   if (rc < 0)
   {
      perror("Niepowodzenie listen()");
      break;
   }

   printf("Gotowy do obsługi klienta (connect()).\n");

   /*******************************************************************/
   /* Serwer używa funkcji accept() do zaakceptowania połączenia       */
   /* przychodzącego. Wywołanie funkcji accept() zostanie zablokowane  */
   /* na nieokreślony czas oczekiwania na połączenie przychodzące.     */
   /*******************************************************************/
```

```
   sd2 = accept(sd, NULL, NULL);
   if (sd2 < 0)
   {
      perror("Niepowodzenie accept()");
      break;
   }

   /********************************************************************/
   /* W tym przykładzie wiadomo, że klient wyśle 250 bajtów danych.    */
   /* Dzięki temu można użyć opcji gniazda SO_RCVLOWAT i określić, że  */
   /* funkcja recv() ma nie wychodzić z uśpienia, dopóki nie zostanie  */
   /* odebrane wszystkie 250 bajtów danych.                            */
   /********************************************************************/
   length = BUFFER_LENGTH;
   rc = setsockopt(sd2, SOL_SOCKET, SO_RCVLOWAT,
                                    (char *)&length, sizeof(length));
   if (rc < 0)
   {
      perror("Niepowodzenie setsockopt(SO_RCVLOWAT)");
      break;
   }


   /********************************************************************/
   /* Odebranie 250 bajtów od klienta                                  */
   /********************************************************************/
   rc = recv(sd2, buffer, sizeof(buffer), 0);
   if (rc < 0)
   {
      perror("Niepowodzenie recv()");
      break;
   }

   printf("Otrzymano dane, bajtów: %d\n", rc);
   if (rc == 0 ||
       rc < sizeof(buffer))
   {
      printf("Klient zamknął połączenie przed wysłaniem\n");
      printf("wszystkich danych\n");
      break;
   }

   /********************************************************************/
   /* Odesłanie danych do klienta                */
   /********************************************************************/
   rc = send(sd2, buffer, sizeof(buffer), 0);
   if (rc < 0)
   {
      perror("Niepowodzenie send()");
      break;
   }

   /********************************************************************/
   /* Zakończenie programu                                             */
   /********************************************************************/

} while (FALSE);

/***********************************************************************/
/* Zamknięcie wszystkich otwartych deskryptorów gniazd                 */
/***********************************************************************/
if (sd != -1)
   close(sd);

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

```
   /********************************************************************/
   /* Usunięcie ścieżki UNIX z systemu plików.                         */
   /********************************************************************/
   unlink(SERVER_PATH);
}
```

Przykład: aplikacja klienta używająca rodziny adresów AF_UNIX_CCSID: Poniższe programy przykładowe używają rodziny adresów AF_UNIX_CCSID. Informacje dotyczące wykorzystania przykładowych kodów zawiera sekcja Informacje dotyczące kodu.

```
/**************************************************************************/
/* Ten przykładowy kod jest aplikacją klienta, używającą rodziny adresów  */
/* AF_UNIX_CCSID.                                                         */
/**************************************************************************/

/**************************************************************************/
/* Wymagane przez program pliki nagłówkowe.                               */
/**************************************************************************/
#include <stdio.h>
#include <string.h>
#include <wcstr.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/unc.h>

/**************************************************************************/
/* Stałe używane przez ten program                                        */
/**************************************************************************/
#define SERVER_PATH     "/tmp/server"
#define BUFFER_LENGTH    250
#define FALSE              0

/* Przekaż 1 parametr, który albo jest     */
/* nazwą ścieżki serwera w postaci ciągu   */
/* znaków UNICODE, albo ustawioną w makrze */
/* #define SERVER_PATH ścieżką serwera,    */
/* która jest łańcuchem CCSID 500.         */
void main(int argc, char *argv[])
{
   /***********************************************************************/
   /* Definicje zmiennych i struktur.                                     */
   /***********************************************************************/
   int    sd=-1, rc, bytesReceived;
   char   buffer[BUFFER_LENGTH];
   struct sockaddr_unc serveraddr;

   /***********************************************************************/
   /* Pętla do/while(FALSE) ułatwia realizację procedur czyszczących w    */
   /* przypadku błędu. Funkcja close() dla deskryptora gniazda jest       */
   /* uruchamiana jednokrotnie na samym końcu programu.                   */
   /***********************************************************************/
   do
   {
      /********************************************************************/
      /* Funkcja socket() zwraca deskryptor gniazda stanowiącego punkt    */
      /* końcowy. Instrukcja ta określa także, że dla dla tego gniazda    */
      /* użyta zostanie rodzina adresów UNIX_CCSID ze strumieniowym       */
      /* protokołem transportowym (SOCK_DGRAM).                           */
      /********************************************************************/
      sd = socket(AF_UNIX_CCSID, SOCK_STREAM, 0);
      if (sd < 0)
     {
         perror("Niepowodzenie socket()");
         break;
      }

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

```
/* Jeśli został przekazany argument, należy go użyć jako nazwy      */
/* serwera, w przeciwnym razie należy użyć zmiennej określonej      */
/* w makrze #define znajdującym się na początku programu.           */
/********************************************************************/
memset(&serveraddr, 0, sizeof(serveraddr));
serveraddr.sunc_family = AF_UNIX_CCSID;
if (argc > 1)
{
   /* Argument jest nazwą ścieżki w kodzie UNICODE.  Należy użyć    */
   /* formatu domyślnego.                                           */
   serveraddr.sunc_format = SO_UNC_DEFAULT;
   wcscpy(serveraddr.sunc_path.wide, (wchar_t *) argv[1]);
}
else
{
   /* Lokalna zmienna #define używa CCSID 500. Ustaw zmienną        */
   /* Qlg_Path_Name, aby został użyty format znakowy.               */
   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);
}
/********************************************************************/
/* Aby nawiązać połączenie z serwerem, zostanie użyta funkcja       */
/* connect().                                                       */
/********************************************************************/
rc = connect(sd, (struct sockaddr *)&serveraddr, sizeof(serveraddr));
if (rc < 0)
{
   perror("Niepowodzenie connect()");
   break;
}

/********************************************************************/
/* Wysłanie 250 bajtów znaków 'a' do serwera                        */
/********************************************************************/
memset(buffer, 'a', sizeof(buffer));
rc = send(sd, buffer, sizeof(buffer), 0);
if (rc < 0)
{
   perror("Niepowodzenie send()");
   break;
}

/********************************************************************/
/* W tym przykładzie wiadomo, że serwer odpowie wysłaniem tych      */
/* samych 250 bajtów, które wysłaliśmy. Ponieważ wiadomo, że        */
/* zostanie odesłanych 250 bajtów, można użyć opcji gniazda         */
/* SO_RCVLOWAT, uruchomić pojedynczą funkcję recv() i pobrać        */
/* wszystkie dane.                                                  */
/*                                                                  */
/* Użycie opcji SO_RCVLOWAT zostało już pokazane w przykładzie      */
/* serwera, dlatego tutaj użyjemy innej metody. Ponieważ te 250     */
/* bajtów danych może być przysłanych w oddzielnych pakietach,      */
/* będziemy wielokrotnie uruchamiali funkcję recv(), dopóki         */
/* nie odbierzemy wszystkich 250 bajtów.                            */
/********************************************************************/
bytesReceived = 0;
while (bytesReceived < BUFFER_LENGTH)
{
   rc = recv(sd, & buffer[bytesReceived],
             BUFFER_LENGTH - bytesReceived, 0);
   if (rc < 0)
   {
      perror("Niepowodzenie recv()");
      break;
```

```
         }
         else if (rc == 0)
         {
            printf("Serwer zamknął połączenie\n");
            break;
         }

         /*****************************************************************/
         /* Zwiększenie liczby otrzymanych dotychczas bajtów              */
         /*****************************************************************/
         bytesReceived += rc;
      }

   } while (FALSE);

   /***********************************************************************/
   /* Zamknięcie wszystkich otwartych deskryptorów gniazd                 */
   /***********************************************************************/
   if (sd != -1)
      close(sd);
}
```

Używanie rodziny adresów AF_TELEPHONY

Rodzina adresów telefonicznych (gniazda korzystające z rodziny adresów AF_TELEPHONY) pozwala użytkownikowi nawiązać (wybrać numer) i odebrać (podnieść słuchawkę) połączenia telefoniczne przez przyłączoną sieć telefonii ISDN za pomocą standardowych funkcji API gniazd. Gniazda tworzące punkty końcowe połączenia w tej domenie w rzeczywistości są wywoływanymi (pasywny punkt końcowy) i wywołującymi (aktywny punkt końcowy) elementami połączenia telefonicznego. Adresy AF_TELEPHONY to numery telefonów składające się z maksymalnie 40 cyfr (0 - 9), które są zawarte w strukturach adresów sockaddr_tel.

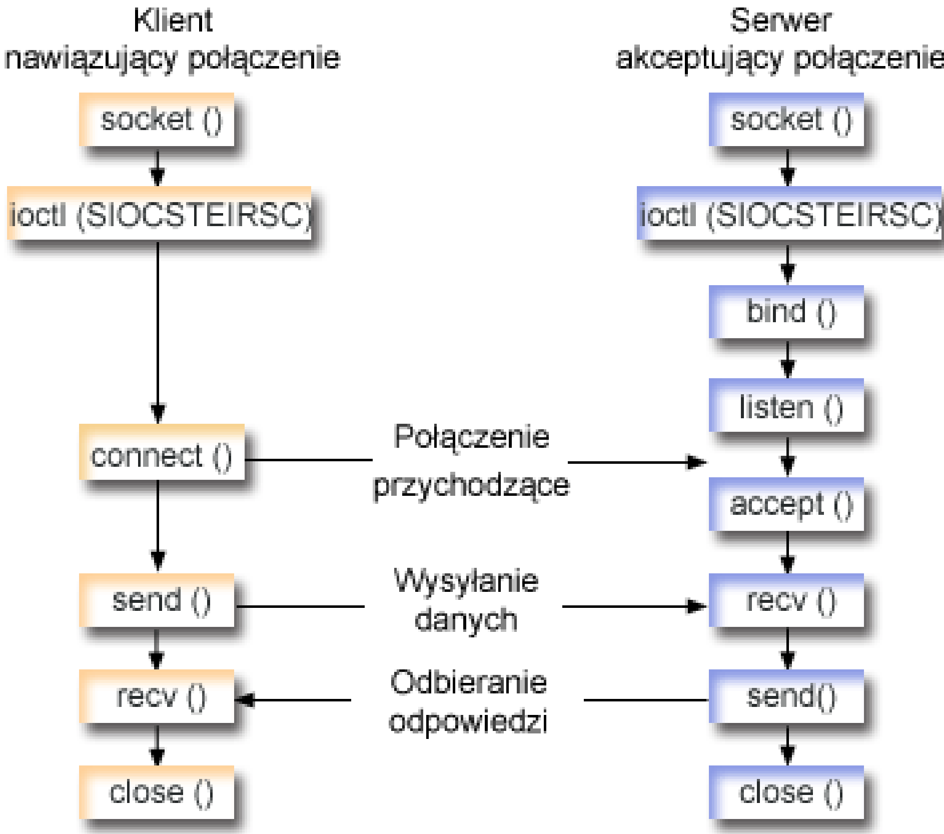
System obsługuje gniazda AF_TELEPHONY tylko jako gniazda zorientowane na połączenie (typ SOCK_STREAM). Semantyka i funkcje tych gniazd są podobne, jak dla innych protokołów zorientowanych na połączenie. Zasadnicza różnica polega na tym, że połączenie w domenie telefonicznej nie zapewnia większej niezawodności niż takie samo połączenie telefoniczne. Jeśli wymagana jest gwarancja dostarczenia informacji, należy osiągnąć to na poziomie aplikacji, na przykład za pomocą aplikacji FAX, korzystających z tej rodziny. W przypadku rodziny adresów telefonicznych nie jest również obsługiwana koncepcja danych spoza pasma.

Przed zainicjowaniem lub odebraniem połączenia gniazda AF_TELEPHONY muszą być powiązane z urządzeniem telefonii sieciowej (aparatem telefonicznym). Aby dokonać tego powiązania, należy użyć specjalnej komendy **SIOCSTELRSC** (ustawienie zasobów telefonicznych) funkcji **ioctl()**. Przed zastosowaniem tej komendy należy skonfigurować te urządzenia i przygotować je do użycia.

Przed wywołaniem SIOCSTELRSC **ioctl()** aplikacja musi rozstrzygnąć nazwę urządzenia. Nazwa ta musi być przetłumaczona na wskaźnik systemowy, którego należy użyć jako danych wejściowych dla komendy SIOCSTELRSC.

Urządzenie pozostanie powiązane z gniazdem do momentu zamknięcia gniazda. W rezultacie z gniazdem można powiązać więcej niż jedno urządzenie. Takie wielokrotne powiązanie umożliwia aplikacji nasłuchiwanie i odpowiadanie na połączenia na więcej niż jednym urządzeniu poprzez pojedyncze gniazdo.

Poniższa ilustracja przedstawia powiązania wywołań funkcji gniazd używanych w odniesieniu do rodziny adresów AF_TELEPHONY. Sekcja Wymagania wstępne do programowania z użyciem gniazd zawiera szczegółowe informacje dotyczące konfigurowania środowiska do używania rodziny adresów AF_UNIX.



Przebieg zdarzeń w gnieździe: klient używający rodziny adresów AF_TELEPHONY
W sekcji Przykład: nawiązywanie połączenia AF_TELEPHONY użyto wywołań następujących funkcji:

1. Funkcja **socket()** zwraca deskryptor gniazda odpowiadający punktowi końcowemu.
2. Do gniazda należy przypisać urządzenie, tłumacząc nazwę urządzenia na wskaźnik systemowy i wypełniając strukturę żądania. Aby powiązać te sobą urządzenie i wskaźnik systemowy, należy użyć funkcji **ioctl()**.
3. Po odebraniu deskryptora gniazda należy użyć funkcji **connect()** do nawiązania połączenia z serwerem.
4. Funkcja **send()** wysyła zawartość buforu wysyłania do klienta.
5. Funkcja **recv()** odbiera dane z aplikacji klienta.
6. Funkcja **close()** zamyka wszystkie otwarte deskryptory gniazd.

Przebieg zdarzeń w gnieździe: aplikacja serwera używająca rodziny adresów AF_TELEPHONY
W sekcji Przykład: przyjmowanie połączenia AF_TELEPHONY użyto wywołań następujących funkcji:

1. Funkcja **socket()** zwraca deskryptor gniazda odpowiadający punktowi końcowemu.
2. Do gniazda należy przypisać urządzenie, tłumacząc nazwę urządzenia na wskaźnik systemowy i wypełniając strukturę żądania. Aby powiązać te sobą urządzenie i wskaźnik systemowy, należy użyć funkcji **ioctl()**.
3. Po odebraniu deskryptora gniazda należy użyć funkcji **connect()** do nawiązania połączenia z serwerem.
4. Funkcja **listen()** umożliwia serwerowi przyjęcie połączenia przychodzącego od klienta.
5. Serwer używa funkcji **accept()** do zaakceptowania żądania połączenia przychodzącego.
6. Funkcja **recv()** odbiera dane z aplikacji klienta.
7. Funkcja **send()** wysyła zawartość buforu wysyłania do klienta.
8. Funkcja **close()** zamyka wszystkie otwarte deskryptory gniazd.

Przykład: nawiązywanie połączenia AF_TELEPHONY: Programy mogą się między sobą komunikować poprzez gniazdo domeny telefonicznej. Poniższy kod można wykorzystać do udostępniania gniazda podczas nawiązywania połączenia z klientem. Informacje dotyczące wykorzystania przykładowych kodów zawiera sekcja Informacje dotyczące kodu.

```
/**************************************************************************/
/* Jest to przykład programu do nawiązywania połączeń AF_TELEPHONY.       */
/**************************************************************************/

#include <stdio.h>                    /* Funkcje łańcuchowe        */
#include <string.h>                   /* Funkcje łańcuchowe        */
#include <miptrnam.h>                 /* Typy wskaźników           */

#include <sys/socket.h>               /* Gniazda                   */
#include <nettel/tel.h>               /* Rodzina adresów telefon.  */
#include <errno.h>                    /* Kody błędów               */
#include <sys/ioctl.h>                /* Kody błędów               */

int main() {
  /******************************************************************/
  /* Różne deklaracje                                               */
  /******************************************************************/
  int xSock, xRC, xLength;

  /******************************************************************/
  /* Tłumaczenie nazwy urządzenia na wskaźnik systemowy             */
  /******************************************************************/
  _SYSPTR pDev;                       /* Wskaźnik systemowy do      */
                                      /* urządzenia                 */
  _RSLV_Template_T xTemp;             /* Szablon do przetłumaczenia */
  char pName[]="FRED                           "; /* Nazwa urządzenia*/
  struct TelResource xResource;       /* struktura SIOCSTELRSC      */

  /******************************************************************/
  /* Struktura adresu gniazda                                       */
  /******************************************************************/
  struct sockaddr_tel xAddr;

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

  /******************************************************************/
  /* Otwarcie gniazda                                               */
  /******************************************************************/
  xSock = socket(AF_TELEPHONY,SOCK_STREAM,0);
  if (xSock<0) {
   perror("Niepowodzenie socket()");
   return(-1);
  }

  /******************************************************************/
  /* Przypisanie gniazda do urządzenia                              */
  /* ...przetłumaczenie nazwy urządzenia na wskaźnik systemowy      */
  /* ...wpisanie struktury tego żądania                             */
  /* ...wysłanie ioctl do przeprowadzenia powiązania                */
  /******************************************************************/
  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("Niepowodzenie ioctl()");
    close(xSock);
    return(-1);
  }

  /*****************************************************************/
  /* Połączenie ze zdalnym zasobem (wybranie numeru wywołania)      */
  /*****************************************************************/
  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("Niepowodzenie connect()");
    close(xSock);
    return(-1);
  }

  /*****************************************************************/
  /* Wysłanie zawartości buforu wysyłania                          */
  /*****************************************************************/
  xRC=send(xSock,pSendBuffer,1024,0);
  if (xRC<0) {
    perror("Niepowodzenie send()");
    close(xSock);
    return(-1);
  }

  /*****************************************************************/
  /* Odebranie odpowiedzi                                          */
  /*****************************************************************/
  xRC=recv(xSock,pRecvBuffer,1024,0);
  if (xRC<0) {
    perror("Niepowodzenie recv()");
    close(xSock);
    return(-1);
   }

  /*****************************************************************/
  /* Wykonane, zamknięcie połączenia i powrót                      */
  /*****************************************************************/
  close(xSock);
  return(0);
}
```

Przykład: akceptowanie połączenia AF_TELEPHONY: Rodzina adresów AF_TELEPHONY jest używana w aplikacjach, które do identyfikacji gniazda korzystają z numerów telefonicznych. Są to głównie aplikacje służące do faksowania. Programy mogą się między sobą komunikować poprzez gniazdo domeny telefonicznej. Poniższy kod można wykorzystać do udostępniania gniazda w celu zaakceptowania wywołania serwera. Informacje dotyczące wykorzystania przykładowych kodów zawiera sekcja Informacje dotyczące kodu.

```
/**********************************************************************/
/* Jest to przykład programu do przyjmowania połączeń AF_TELEPHONY.      */
/**********************************************************************/
#include <stdio.h>                  /* Funkcje łańcuchowe        */
#include <string.h>                 /* Funkcje łańcuchowe        */
#include <miptrnam.h>               /* Typy wskaźników           */

#include <sys/socket.h>             /* Gniazda                   */
#include <nettel/tel.h>             /* Rodzina adresów telefon.   */
#include <errno.h>                  /* Kody błędów               */
```

```
#include <sys/ioctl.h>              /* Kody błędów              */

int main() {

  /*******************************************************************/
  /* Różne deklaracje                                                */
  /*******************************************************************/
  int xSock,xNewSock,xRC,xLength;

  /****************************************************************/
  /* Tłumaczenie nazwy urządzenia na wskaźnik systemowy           */
  /****************************************************************/
  _SYSPTR pDev;                     /* Wskaźnik systemowy do      */
                                    /* urządzenia                 */
  _RSLV_Template_T xTemp;           /* Szablon do przetłumaczenia */
  char pName[]="GEORGE                       "; /*Nazwa urządzenia*/
  struct TelResource xResource;     /* struktura SIOCSTELRSC      */

  /****************************************************************/
  /* Struktura adresu gniazda                                     */
  /****************************************************************/
  struct sockaddr_tel xAddr;

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

  /*****************************************************************/
  /* Otwarcie gniazda                                              */
  /*****************************************************************/
  xSock = socket(AF_TELEPHONY,SOCK_STREAM,0);
  if (xSock<0) {
    perror("Niepowodzenie socket()");
    return(-1);
  }

  /****************************************************************/
  /*...najpierw, przetłumacz nazwę urządzenia na wskaźnik systemowy*/
  /*...potem, wpisz strukturę tego żądania                        */
  /*...na koniec, wyślij ioctl w celu przeprowadzenia powiązania   */
  /****************************************************************/
  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("Niepowodzenie ioctl()");
    close(xSock);
    return(-1);
  }

  /****************************************************************/
  /* Nawiązanie do numeru lokalnego (użycie TELADDR_ANY oznacza    */
  /* zaakceptowanie wywołań dla dowolnego numeru znajdującego się  */
  /* na liście połączeń przychodzących)                            */
  /****************************************************************/
  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("Niepowodzenie bind()");
    close(xSock);
    return(-1);
  }

  /*****************************************************************/
  /* Nasłuchiwanie połączeń przychodzących                        */
  /*****************************************************************/
  xRC=listen(xSock,5);
  if (xRC<0) {
    perror("Niepowodzenie listen()");
    close(xSock);
    return(-1);
  }
  /*****************************************************************/
  /* Zaakceptowanie połączenia przychodzącego                     */
  /*****************************************************************/
  memset(&xAddr,0x00,sizeof(xAddr));
  xLength = sizeof(xAddr);
  xNewSock=accept(xSock,(struct sockaddr*)&xAddr,&xLength);
  if (xNewSock<0) {
    perror("Niepowodzenie accept()");
    close(xSock);
    return(-1);
  }

  /*****************************************************************/
  /* Przyjęcie danych                                              */
  /*****************************************************************/
  xRC=recv(xNewSock,pRecvBuffer,1024,0);
  if (xRC<0) {
    perror("Niepowodzenie recv()");
    close(xSock);
    close(xNewSock);
    return(-1);
  }

  /*****************************************************************/
  /* Wysłanie odpowiedzi                                           */
  /*****************************************************************/
  xRC=send(xNewSock,pSendBuffer,1024,0);
  if (xRC<0) {
    perror("Niepowodzenie send()");
    close(xSock);
    close(xNewSock);
    return(-1);
  }

  /*****************************************************************/
  /* Wykonane, zamknięcie połączenia i powrót                     */
  /*****************************************************************/
  close(xSock);
  close(xNewSock);
  return(0);
}
```

Pojęcia dotyczące gniazd

W poniższych sekcjach omówiono zaawansowane koncepcje dotyczące gniazd, które wychodzą poza zakres ogólnych wiadomości o tym, czym są gniazda i jak działają. W sekcjach tych przedstawiono metody projektowania aplikacji używających gniazd dla większych i bardziej złożonych sieci. Każda z wymienionych poniżej sekcji zawiera odpowiedni program przykładowy.

- Asynchroniczne operacje we/wy
- Gniazda chronione
- Obsługa klienta SOCKS
- Ochrona wątków
- Nieblokujące operacje we/wy
- Sygnały
- Rozsyłanie grupowe IP
- Przesyłanie danych pliku – funkcje send_file() accept_and_recv()
- Dane spoza pasma
- Multipleksowanie operacji we/wy – funkcja select()
- Funkcje sieciowe gniazd
- Obsługa systemu nazw domen (DNS)
- Zgodność z BSD
- Przekazywanie deskryptorów pomiędzy procesami –funkcje sendmsg() i recvmsg()

Asynchroniczne operacje we/wy

Funkcje API asynchronicznych operacji we/wy udostępniają metodę dla wątków modelu klient serwer, umożliwiającą realizację współbieżnych operacji we/wy z efektywnym wykorzystaniem pamięci. W poprzednich wątkach modelu klient/serwer przeważały dwa modele operacji we/wy. W pierwszym z nich jednemu połączeniu klienta dedykowano jeden wątek. Prowadziło to do używania zbyt wielu wątków i mogło powodować dodatkowe obciążenie związane z przejściem w stan nieaktywny i uaktywnieniem. W drugim modelu minimalizuje się liczbę wątków za pomocą funkcji API **select()** dla dużego zestawu połączeń klientów i delegując przygotowane połączenia klientów lub żądanie do wątku. W tym modelu trzeba wybierać lub zaznaczać wyniki wcześniejszego wyboru, co wymaga wykonania znacznej i powtarzalnej pracy.

Asynchroniczne operacje we/wy i nakładanie operacji we/wy eliminują oba te problemy, przekazując dane z i do buforu użytkownika po przekazaniu sterowania do aplikacji użytkownika. Asynchroniczne operacje we/wy powiadamiają wątki procesów roboczych o dostępności danych do odczytu oraz o gotowości połączenia do transmisji danych.

Zalety asynchronicznych operacji we/wy

- Efektywniejsze wykorzystanie zasobów systemu.
  Kopiowanie danych z i do buforu użytkownika odbywa się asynchronicznie względem aplikacji inicjującej żądanie. To nakładające się przetwarzanie umożliwia efektywne wykorzystanie wielu procesorów i w wielu przypadkach poprawia szybkość stronicowania, ponieważ bufory systemowe są zwalniane do ponownego wykorzystania, kiedy nadsyłane są dane.
- Minimalizacja czasu oczekiwania dla procesu/wątku.
- Natychmiastowe udostępnienie usługi na żądanie klienta.
- Zmniejszenie średniego obciążenia związanego z uśpieniem i uaktywnieniem systemu.
- Wydajna obsługa ″aplikacji impulsowych″.
- Lepsza skalowalność.
- Najbardziej efektywne metody obsługi przesyłania danych o dużej objętości.
  Opcja fillBuffer funkcji API **QsoStartRecv()** informuje system operacyjny o uzyskaniu dużych ilości danych przed zakończeniem asynchronicznych operacji we/wy. Duże ilości danych można również przesyłać w ramach jednej operacji asynchronicznej.
- Minimalizacja liczby potrzebnych wątków.
- Możliwość użycia zegarów w celu określenia maksymalnej ilości czasu wymaganej do asynchronicznego zakończenia tej operacji. Jeśli połączenie z klientem było bezczynne przez ustalony okres czasu, serwery zamkną je. Zegary asynchroniczne umożliwiają serwerowi wymuszenie tego limitu czasu.
- Asynchroniczne zainicjowanie chronionej sesji przy użyciu funkcji API **gsk_secure_soc_startInit()**.

Tabela 13. Funkcje API asynchronicznych operacji we/wy

Funkcja	Opis
gsk_secure_soc_startInit()	Uruchamia asynchroniczną negocjację bezpiecznej sesji, używając zestawu atrybutów dla środowiska SSL i bezpiecznej sesji. **Uwaga:** Ta funkcja API obsługuje tylko gniazda typu SOCK_STREAM z rodziny adresów AF_INET i AF_INET6.
gsk_secure_soc_startRecv()	Rozpoczyna operację asynchronicznego odbioru w ramach bezpiecznej sesji. **Uwaga:** Ta funkcja API obsługuje tylko gniazda typu SOCK_STREAM z rodziny adresów AF_INET i AF_INET6.
gsk_secure_soc_startSend()	Rozpoczyna operację asynchronicznego wysyłania w ramach bezpiecznej sesji. **Uwaga:** Ta funkcja API obsługuje tylko gniazda typu SOCK_STREAM z rodziny adresów AF_INET i AF_INET6.
QsoCreateIOCompletionPort()	Tworzy wspólny punkt oczekiwania dla zakończonych, nakładających się asynchronicznych operacji we/wy. Funkcja **QsoCreateIOCompletionPort()** zwraca uchwyt portu reprezentującego punkt oczekiwania. Uchwyt ten należy podać dla funkcji **QsoStartRecv()**, **QsoStartSend()**, **QsoStartAccept()**, **gsk_secure_soc_startRecv()** lub **gsk_secure_soc_startSend()**, aby zainicjować nakładające się, asynchroniczne operacje we/wy. Ponadto uchwyt ten może zostać użyty z funkcją **QsoPostIOCompletion()** do przesłania zdarzenia do powiązanego portu we/wy zakończenia.
QsoDestroyIOCompletionPort()	Niszczy port we/wy zakończenia.
QsoWaitForIOCompletionPort()	Czeka na zakończone, nakładające się operacje we/wy. Port we/wy zakończenia reprezentuje ten port oczekiwania.
QsoStartAccept()	Rozpoczyna asynchroniczną operację akceptowania. **Uwaga:** Ta funkcja API obsługuje tylko gniazda typu SOCK_STREAM z rodziny adresów AF_INET i AF_INET6.
QsoStartRecv()	Rozpoczyna asynchroniczną operację odbioru. **Uwaga:** Ta funkcja API obsługuje tylko gniazda typu SOCK_STREAM z rodziny adresów AF_INET i AF_INET6.
QsoStartSend()	Rozpoczyna asynchroniczną operację wysyłania. **Uwaga:** Ta funkcja API obsługuje tylko gniazda typu SOCK_STREAM z rodziny adresów AF_INET i AF_INET6.
QsoPostIOCompletion()	Pozwala aplikacji powiadomić port zakończenia o wystąpieniu pewnej funkcji lub czynności.

Jak działają asynchroniczne operacje we/wy

Aplikacja tworzy port we/wy zakończenia za pomocą funkcji API **QsoCreateIOCompletionPort()**. Funkcja ta zwraca uchwyt, którego można użyć do zaplanowania i oczekiwania na zakończenie żądań asynchronicznych operacji we/wy. Następnie aplikacja uruchamia funkcję operacji wejścia lub operacji wyjścia, podając uchwyt portu we/wy zakończenia. Kiedy operacje we/wy zostają zakończone, informacje o statusie i zdefiniowany przez aplikację uchwyt zostają zapisane w podanym porcie we/wy zakończenia. Zapisanie do portu we/wy zakończenia uaktywnia dokładnie jeden z możliwych wielu wątków oczekujących. Aplikacja odbiera:

- bufor dostarczony w pierwotnym żądaniu,
- długość danych przetworzonych z lub do tego buforu,
- wskazanie typu zakończonej operacji we/wy,
- zdefiniowany przez aplikację uchwyt, który został przekazany w początkowym żądaniu operacji we/wy.

Ten uchwyt aplikacji może być po prostu deskryptorem gniazda, wskazującym połączenie klienta, lub wskaźnikiem do pamięci, która zawiera dodatkowe informacje o stanie połączenia klienta. Ponieważ operacja zakończyła się, a uchwyt aplikacji został przekazany, wątek procesu roboczego określa następny krok w celu zakończenia połączenia klienta. Wątki procesów roboczych, które przetwarzają te zakończone operacje asynchroniczne, mogą mieć wiele różnych żądań klienta i nie są powiązane tylko z jednym z nich. Ponieważ kopiowanie z i do buforów użytkowników odbywa się asynchronicznie względem procesów serwera, czasy oczekiwania dla żądań klientów skracają się. Cecha ta może być szczególnie korzystna w systemach wieloprocesorowych.

Przykład prostego modelu serwera, który wykorzystuje asynchroniczne operacje we/wy, znajduje się w sekcji Przykład: korzystanie z asynchronicznych operacji we/wy.

Gniazda chronione

Obecnie system OS/400 obsługuje dwa sposoby tworzenia aplikacji gniazd chronionych w systemie iSeries.Funkcje API SSL_ i Global Secure Toolkit (GSKit) zapewniają prywatność komunikacji w otwartych sieciach komunikacyjnych, czyli najczęściej w Internecie. Funkcje te umożliwiają aplikacjom typu klient/serwer komunikację zabezpieczoną przed podsłuchiwaniem, manipulowaniem i fałszowaniem danych. Obie funkcje obsługują uwierzytelnianie serwera i klienta oraz umożliwiają aplikacji użycie protokołu SSL (Secure Sockets Layer). Jednakże funkcje API GSKit są obsługiwane przez wszystkie platformy @server IBM, a funkcje API SSL_ są rodzime dla systemu operacyjnego OS/400. Aby zapewnić współdziałanie pomiędzy platformami, podczas tworzenia aplikacji obsługujących połączenia gniazd chronionych zaleca się używanie funkcji API GSKiT.

Opis tych funkcji API zawierają następujące tematy:

- Funkcje API Global Secure Toolkit (GSKit)
- Funkcje API SSL_

Temat Kody komunikatów o błędach funkcji API gniazd chronionych zawiera listę najczęściej występujących komunikatów o błędach funkcji API gniazd chronionych.

Przegląd gniazd chronionych

Opracowany przez firmę Netscape Protokół SSL (Secure Sockets Layer) jest protokołem warstwowym, który jest stosowany w połączeniu z niezawodnym protokołem transportowym, takim jak Transmission Control Protocol (TCP), w celu zapewnienia aplikacjom bezpiecznej komunikacji. Przykładami wielu zastosowań, w których potrzebna jest bezpieczna komunikacja, mogą być protokoły HTTPS, FTPS, SMTP i TELNETS.

Aplikacje obsługujące protokół SSL potrzebują zazwyczaj innego portu, niż aplikacje, które tego protokołu nie obsługują. Na przykład przeglądarka z obsługą SSL łączy się z serwerem protokołu Hypertext Transfer Protocol (HTTP) obsługującym SSL za pomocą adresu Universal Resource Locator (URL), który zaczyna się od ″HTTPS″, a nie od ″HTTP″. W większości przypadków taki adres będzie wymagał otwarcia portu 443, a nie portu 80, używanego przez standardowy serwer HTTP.

Istnieje wiele zdefiniowanych wersji protokołu SSL. Najnowsza wersja, Transport Layer Security (TLS) wersja 1.0, stanowi ewolucyjną aktualizację protokołu SSL wersja 3.0. Zarówno rodzime funkcje API systemu iSeries - SSL_ - jak i funkcje API GSKit obsługują protokoły: TLS wersja 1.0, TLS wersja 1.0 kompatybilna z SSL wersja 3.0, SSL wersja 3.0, SSL wersja 2.0 i SSL wersja 3.0 kompatybilna z wersją 2.0. Więcej informacji o protokole TLS Wersja 1.0 zawiera dokument RFC 2246 ″Transport Layer Security″ opracowany przez Internet Engineering Task Force (IETF)

Funkcje API Global Secure ToolKit (GSKit)

GSKit to zestaw programowalnych interfejsów, który umożliwia aplikacjom obsługę warstwy SSL. Podobnie jak funkcje API SSL_, funkcje API GSKit umożliwiają dostęp do funkcji SSL i TLS z poziomu aplikacji używających gniazd. W przeciwieństwie jednak do funkcji API SSL_ są one obsługiwane przez wszystkie platformy @server IBM i są łatwiejsze w programowaniu niż poprzednie funkcje API SSL_. Ponadto dodano nowe funkcje API GSKit w celu utworzenia asynchronicznej instancji sesji gniazd chronionych. Funkcje te zapewniają chronione połączenie

obsługujące wiele klientów oraz sytuacje, gdy liczba żądań przychodzących jest duża i wymaga wielu zadań. Jednak te funkcje API są rodzimymi funkcjami systemu OS/400 i nie można ich przenosić na inne platformy @server.

Uwaga: Te funkcje API obsługują tylko gniazda typu SOCK_STREAM z rodziny adresów AF_INET i AF_INET6.
Poniższa tabela przedstawia funkcje API GSKit:

Tabela 14. Funkcje API Global Secure Toolkit

Funkcja	Opis
gsk_attribute_get_buffer()	Uzyskuje specyficzne informacje w postaci łańcucha znaków o bezpiecznej sesji lub o środowisku SSL, takie jak zbiór bazy certyfikatów, hasło bazy certyfikatów, identyfikator aplikacji i szyfr.
gsk_attribute_get_cert_info()	Uzyskuje specyficzne informacje o certyfikacie serwera lub klienta dla bezpiecznej sesji lub środowiska SSL.
gsk_attribute_get_enum_value()	Uzyskuje wartości specyficznych, wyszczególnionych danych dla bezpiecznej sesji lub dla środowiska SSL.
gsk_attribute_get_numeric_value()	Uzyskuje określone informacje numeryczne o bezpiecznej sesji lub środowisku SSL.
gsk_attribute_set_buffer()	Nadaje określonemu atrybutowi buforu wartość odpowiadającą określonej bezpiecznej sesji lub środowisku SSL.
gsk_attribute_set_enum()	Przypisuje wyszczególniony atrybut typu do wyszczególnionej wartości w bezpiecznej sesji lub środowisku SSL.
gsk_attribute_set_numeric_value()	Przypisuje określone informacje numeryczne bezpiecznej sesji lub środowisku SSL.
gsk_environment_close()	Zamyka środowisko SSL i zwalnia całą pamięć przypisaną środowisku.
gsk_environment_init()	Inicjuje środowisko SSL po ustawieniu wymaganych atrybutów.
gsk_environment_open()	Zwraca uchwyt środowiska SSL, który musi zostać zapisany i użyty podczas następnych wywołań funkcji gsk.
gsk_secure_soc_close()	Zamyka bezpieczną sesję i zwalnia wszystkie przypisane jej zasoby.
gsk_secure_soc_init()	Negocjuje bezpieczną sesję, używając zestawu atrybutów dla środowiska SSL i bezpiecznej sesji.
gsk_secure_soc_misc()	Realizuje różne funkcje bezpiecznej sesji.
gsk_secure_soc_open()	Uzyskuje pamięć dla bezpiecznej sesji, ustawia domyślne wartości atrybutów i zwraca uchwyt, który musi zostać zapisany i użyty podczas następnych wywołań funkcji związanych z bezpieczną sesją.
gsk_secure_soc_read()	Odbiera dane podczas bezpiecznej sesji.
gsk_secure_soc_startInit()	Uruchamia asynchroniczną negocjację bezpiecznej sesji, używając zestawu atrybutów dla środowiska SSL i bezpiecznej sesji.
gsk_secure_soc_write()	Zapisuje dane podczas bezpiecznej sesji.
gsk_secure_soc_startRecv()	Inicjuje asynchroniczną operację odbioru podczas bezpiecznej sesji.
gsk_secure_soc_startSend()	Inicjuje asynchroniczną operację wysyłania podczas bezpiecznej sesji.
gsk_strerror()	Pobiera komunikat o błędzie i powiązany łańcuch tekstowy, który opisuje wartość zwracaną podczas wywołania funkcji API GSK.

Aplikacja, która korzysta z funkcji API gniazd i funkcji API GSKit, zawiera następujące elementy:

1. Wywołanie funkcji **socket()**, aby uzyskać deskryptor gniazda.
2. Wywołanie funkcji **gsk_environment_open()** w celu uzyskania uchwytu środowiska SSL.
3. Przynajmniej jedno wywołanie funkcji gsk_attribute_set_xxxxx() w celu ustawienia atrybutów środowiska SSL. Minimum to wywołanie funkcji **gsk_attribute_set_buffer()** w celu ustawienia wartości GSK_OS400_APPLICATION_ID lub wartości GSK_KEYRING_FILE. Należy ustawić tylko jedną z tych wartości. Zalecane jest użycie wartości GSK_OS400_APPLICATION_ID. Ponadto należy ustawić typ aplikacji (klient lub serwer), GSK_SESSION_TYPE, używając funkcji **gsk_attribute_set_enum()**.
4. Wywołanie funkcji **gsk_environment_init()** w celu zainicjowania tego środowiska do przetwarzania SSL i określenia informacji o ochronie dla wszystkich sesji SSL, które będą uruchamiane w tym środowisku.
5. Wywołania gniazd do uaktywnienia połączenia. Aplikacja wywołuje funkcję **connect()** w celu uaktywnienia połączenia z programem klienta lub funkcje **bind()**, **listen()** i **accept()** , aby umożliwić serwerowi akceptowanie przychodzących żądań połączeń.
6. Wywołanie funkcji **gsk_secure_soc_open()** w celu uzyskania uchwytu dla bezpiecznej sesji.
7. Przynajmniej jedno wywołanie funkcji gsk_attribute_set_xxxxx() w celu ustawienia atrybutów bezpiecznej sesji. Minimum to wywołanie funkcji **gsk_attribute_set_numeric_value()** w celu powiązania określonego gniazda z tą bezpieczną sesją.
8. Wywołanie funkcji **gsk_secure_soc_init()** w celu zainicjowania negocjacji parametrów szyfrowania podczas uzgadniania SSL.

   Uwaga: Aby uzgadnianie SSL się powiodło, program serwera musi okazać certyfikat. Ponadto serwer musi mieć dostęp do klucza prywatnego powiązanego z certyfikatem serwera i zbioru bazy danych kluczy, w którym przechowywany jest certyfikat. W niektórych przypadkach klient także musi okazać certyfikat podczas przetwarzania uzgadniania SSL. Dotyczy to sytuacji, w której serwer, z którym łączy się klient, ma włączone uwierzytelnianie klienta. Wywołania funkcji API **gsk_attribute_set_buffer**(GSK_OS400_APPLICATION_ID) lub **gsk_attribute_set_buffer**(GSK_KEYRING_FILE) identyfikują (różnymi metodami) zbiór bazy danych kluczy, z którego uzyskano klucz prywatny, i certyfikat użyte podczas uzgadniania.
9. Wywołania funkcji **gsk_secure_soc_read()** i **gsk_secure_soc_write()** w celu odbierania i wysyłania danych.
10. Wywołanie funkcji **gsk_secure_soc_close()** w celu zakończenia bezpiecznej sesji.
11. Wywołanie funkcji **gsk_environment_close()** w celu zamknięcia środowiska SSL.
12. Wywołanie funkcji **close()** w celu usunięcia podłączonego gniazda.

Przykładowe programy wykorzystujące funkcje API GSKit znajdują się w następujących sekcjach:

- Przykład: chroniony serwer z asynchronicznym odbieraniem danych
- Przykład: chroniony serwer używający asynchronicznego uzgadniania
- Przykład: chroniony klient używający funkcji API Global Secure ToolKit (GSKit)

Funkcje API SSL_

Funkcje API SSL_ umożliwiają tworzenie aplikacji gniazd chronionych w systemie iSeries. W przeciwieństwie do funkcji API GSKit, funkcje API SSL_ są rodzime w systemie OS/400.W poniższej tabeli opisano dziewięć funkcji API SSL_ obsługiwanych w implementacji systemu OS/400. Aby dokładnie poznać poszczególne funkcje API, należy użyć odsyłaczy do wykazu funkcji API w Centrum informacyjnym.

*Tabela 15. Funkcje API SSL_*

Funkcja	Opis
SSL_Create()	Umożliwia obsługę SSL dla określonego deskryptora gniazda.
SSL_Destroy()	Kończy obsługę SSL dla określonej sesji i gniazda SSL.
SSL_Handshake()	Inicjuje protokół uzgadniania SSL.

Tabela 15. Funkcje API SSL_ (kontynuacja)

SSL_Init()	Inicjuje bieżące zadanie dla SSL i określa informacje o ochronie dla bieżącego zadania. **Uwaga:** Aby użyć protokołu SSL, wcześniej należy uruchomić funkcję API **SSL_Init()** lub **SSL_Init_Application()**.
SSL_Init_Application()	Inicjuje bieżące zadanie dla SSL i określa informacje o ochronie dla bieżącego zadania. **Uwaga:** Aby użyć protokołu SSL, wcześniej należy uruchomić funkcję API **SSL_Init()** lub **SSL_Init_Application()**.
SSL_Read()	Odbiera dane z deskryptora gniazda z obsługą SSL.
SSL_Write()	Zapisuje dane do deskryptora gniazda z obsługą SSL.
SSL_Strerror()	Odtwarza komunikaty o błędach wykonania SSL.
SSL_Perror()	Drukuje komunikaty o błędach SSL.

Aplikacja, która korzysta z gniazd i funkcji API SSL_, zawiera następujące elementy:

- Wywołanie funkcji **socket()**, aby uzyskać deskryptor gniazda.
- Wywołanie funkcji **SSL_Init()** lub **SSL_Init_Application()** w celu zainicjowania środowiska pracy dla przetwarzania SSL i w celu określenia informacji o ochronie SSL dla wszystkich sesji SSL, które będą uruchamiane w bieżącym zadaniu. Należy użyć tylko jednej z tych funkcji API. Zalecane jest użycie funkcji API **SSL_Init_Application()**.
- Wywołania gniazd do uaktywnienia połączenia. Aplikacja wywołuje funkcję **connect()** w celu uaktywnienia połączenia z programem klienta lub funkcje **bind()**, **listen()** i **accept()**, aby serwer mógł zaakceptować przychodzące żądania połączenia.
- Wywołanie funkcji **SSL_Create()**, aby umożliwić obsługę SSL dla połączonych gniazd.
- Wywołanie funkcji **SSL_Handshake()**, aby zainicjować negocjację parametrów szyfrowania.

  Uwaga: Aby uzgadnianie SSL się powiodło, program serwera musi okazać certyfikat. Ponadto serwer musi mieć dostęp do klucza prywatnego powiązanego z certyfikatem serwera i zbioru bazy danych kluczy, w którym przechowywany jest certyfikat. W niektórych przypadkach klient także musi okazać certyfikat podczas przetwarzania uzgadniania SSL. Dotyczy to sytuacji, w której serwer, z którym łączy się klient, ma włączone uwierzytelnianie klienta. Funkcje API **SSL_Init()** i **SSL_Init_Application()** identyfikują (choć w różny sposób) zbiór bazy danych, z którego pobierany jest certyfikat, i klucz prywatny, używane podczas uzgadniania.
- Wywołania funkcji **SSL_Read()** i **SSL_Write()**, aby otrzymywać i wysyłać dane.
- Wywołanie funkcji **SSL_Destroy()**, aby wyłączyć obsługę SSL dla gniazda.
- Wywołanie funkcji **close()**, aby usunąć połączone gniazda.

Przykładowe programy wykorzystujące te funkcje API SSL_ znajdują się w następujących sekcjach:

- Przykład: chroniony serwer używający funkcji API SSL_
- Przykład: chroniony klient używający funkcji API SSL_

Komunikaty z kodami błędów funkcji API gniazd chronionych

Aby uzyskać informacje dotyczące opisanych poniżej komunikatów o kodach błędów:

1. W wierszu komend wpisz:
   ```
   DSPMSGD RANGE(XXXXXXX)
   ```
   gdzie XXXXXXX jest identyfikatorem komunikatu dla danego kodu powrotu. Na przykład, jeśli kod powrotu był równy 3, wpisz:
   ```
   DSPMSGD RANGE(CPDBCB9)
   ```
2. Wybierz **1**, aby wyświetlić tekst komunikatu.

Tabela 16. Komunikaty o kodach błędów funkcji API gniazd chronionych

Kod powrotu	ID komunikatu	Nazwa stałej
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
6001	CPDBCB1	GSK_OS400_ERROR_PASSWORD_EXPIRED
6002	CPDBCC9	GSK_OS400_ERROR_NOT_REGISTERED
6003	CPDBCAD	GSK_OS400_ERROR_NO_ACCESS

Tabela 16. Komunikaty o kodach błędów funkcji API gniazd chronionych (kontynuacja)

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	Pomyślny powrót
-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 (odwzoruj na -4)
-16	CPDBC8B	SSL_ERROR_BAD_PEER (odwzoruj na -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
-98	CPDBCB8	SSL_ERROR_CLOSED
-99	CPDBCB9	SSL_ERROR_UNKNOWN
-1009	CPDBCC9	SSL_ERROR_NOT_REGISTERED

Tabela 16. Komunikaty o kodach błędów funkcji API gniazd chronionych (kontynuacja)

-1011	CPDBCCB	SSL_ERROR_NO_CERTIFICATE_AUTHORITIES
-9998	CPDBCD8	SSL_ERROR_NO_REUSE

Obsługa klienta SOCKS

W serwerze iSeries zastosowano mechanizm SOCKS wersja 4, aby umożliwić programom, które używają rodziny adresów AF_INET z gniazdami typu SOCK_STREAM, komunikowanie się z programami serwera działającymi w systemach poza firewallem. Firewall to bardzo bezpieczny host, umieszczany pomiędzy chronioną siecią wewnętrzną a niechronioną siecią zewnętrzną. Zazwyczaj taka konfiguracja sieci nie umożliwia komunikacji pomiędzy chronionym hostem a siecią niechronioną. Serwery proxy znajdujące się na firewallu pomagają zarządzać niezbędną komunikacją pomiędzy chronionymi hostami a sieciami niechronionymi.

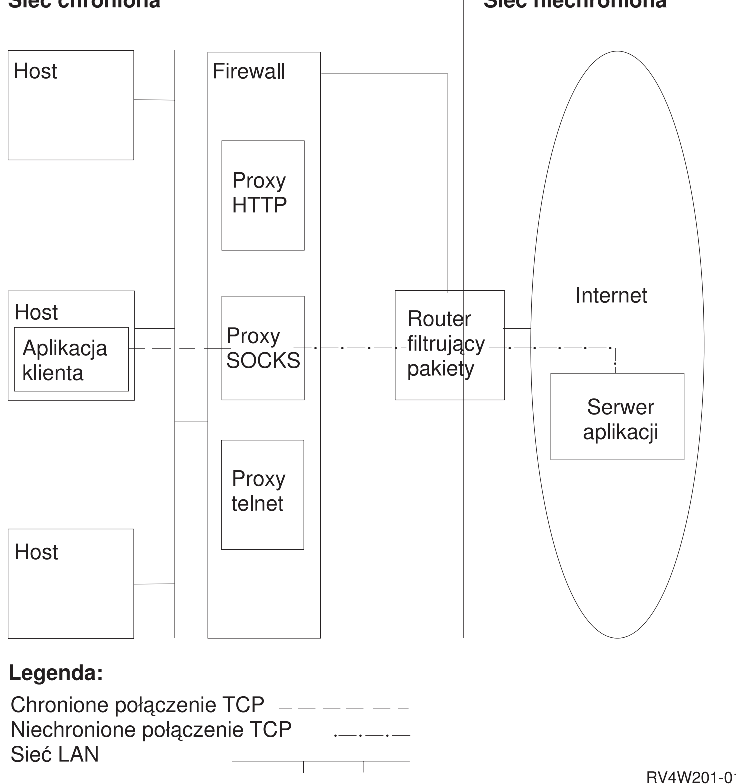
Aplikacje działające na serwerze w chronionej sieci wewnętrznej muszą wysyłać swoje zgłoszenia przez serwery proxy firewall. Serwery te mogą następnie przekazać zgłoszenia do serwera istniejącego w mniej chronionej sieci. Dodatkowo mogą przekazać odpowiedź do aplikacji serwera rozpoczynającego komunikację. Najczęstszym przykładem serwera proxy jest serwer proxy HTTP. Serwery proxy wykonują następujące zadania dla klientów HTTP:

- ukrywają sieć wewnętrzną przed systemami z zewnątrz,
- chronią host przed dostępem bezpośrednim z systemów zewnętrznych,
- odpowiednio zaprojektowane i skonfigurowane mogą filtrować dane przychodzące z zewnątrz.

Serwer proxy HTTP obsługuje tylko klientów HTTP.

Najczęściej spotykaną alternatywą uruchamiania wielu serwerów proxy na firewall jest uruchomienie bardziej wydajnego serwera proxy, znanego jako serwer SOCKS. Może on działać jako pośrednik dla dowolnego połączenia klienta TCP ustanawianego za pomocą API gniazd. Główną korzyścią z obsługi klienta SOCKS iSeries jest umożliwienie aplikacjom klienta przezroczystego dostępu do serwera SOCKS, nie wymagającego zmian w kodzie klienta.

Poniższy rysunek przedstawia typowy projekt firewall z serwerami proxy HTTP, proxy telnet i proxy SOCKS. Należy zauważyć, że do ochrony dostępu klienta do serwera w Internecie używane są dwa oddzielne połączenia TCP. Jedno połączenie wiedzie z chronionego hosta do serwera SOCKS, drugie natomiast z sieci niechronionej do serwera SOCKS.



Aby korzystać z serwera SOCKS, należy na chronionym hoście klienta wykonać dwie czynności:

1. Skonfigurować serwer SOCKS. 15 lutego 2000 IBM poinformował, że produkt IBM Firewall for iSeries (5769-FW1), który obsługuje serwer SOCKS, nie będzie modyfikowany poza swoje obecne możliwości w wersji V4R4.
2. W chronionym systemie klienta należy zdefiniować wszystkie wychodzące połączenia TCP, które mają być skierowane do serwera SOCKS w systemie klienta. Pozycje konfiguracji klienta SOCKS można definiować za pomocą zakładki SOCKS w programie iSeries Navigator w produkcie iSeries Access 95 lub Windows NT. Zakładka SOCKS zawiera zasadniczą pomoc dotyczącą konfigurowania systemu chronionego klienta do obsługi klienta SOCKS.

   Aby skonfigurować obsługę klienta SOCKS, wykonaj poniższe czynności:

   a. W programie iSeries Navigator rozwiń **Serwer iSeries --> Sieć --> Konfiguracja TCP/IP**.
   b. Kliknij prawym klawiszem myszy **Konfiguracja TCP/IP**.
   c. Kliknij **Właściwości**.
   d. Kliknij zakładkę **SOCKS**.
   e. Wpisz informacje o połączeniu na stronie SOCKS.

   Uwaga: System zapisze dane o konfiguracji chronionego klienta SOCKS w zbiorze QASOSCFG w bibliotece QUSRSYS w systemie hosta chronionego klienta.

Po skonfigurowaniu system automatycznie przekieruje ustalone połączenia wychodzące do serwera SOCKS podanego na zakładce SOCKS. Nie trzeba wprowadzać żadnych zmian do aplikacji chronionego klienta. Po odebraniu żądania, serwer SOCKS ustanowi oddzielne połączenie zewnętrzne TCP/IP do serwera w mniej chronionej sieci. Następnie serwer SOCKS przekaże dane pomiędzy wewnętrznym i zewnętrznym połączeniem TCP/IP.

Uwaga: Zdalny host w mniej chronionej sieci łączy się bezpośrednio z serwerem SOCKS. Nie ma bezpośredniego dostępu do chronionego klienta.

Do tego miejsca omawiane były tylko wychodzące połączenia TCP inicjowane przez chronionego klienta. Obsługa

klienta SOCKS umożliwia także akceptację przez serwer SOCKS żądań połączenia przychodzących poprzez firewall. Komunikację tę umożliwia wywołanie funkcji **Rbind()** z systemu chronionego klienta. Aby funkcja **Rbind()** mogła działać, chroniony klient musi wcześniej wywołać funkcję **connect()**, dzięki któremu zostanie utworzone połączenie wychodzące przez serwer SOCKS. Połączenie przychodzące **Rbind()** musi pochodzić spod tego samego adresu IP, który był użyty w połączeniu wychodzącym nawiązanym przez funkcję **connect()**.

Poniższy rysunek przedstawia szczegółowy przegląd interakcji funkcji gniazd z serwerem SOCKS w sposób przezroczysty dla aplikacji. W przykładzie tym klient FTP wywołuje funkcję **Rbind()** zamiast funkcji **bind()**[1]. Wywołuje ją dzięki ponownej kompilacji kodu klienta FTP z definicją preprocesora (#define) __Rbind, która definiuje funkcję **bind()** jako **Rbind()**. Alternatywnie aplikacja może mieć jawnie zakodowane **Rbind()** w odpowiednim kodzie źródłowym. Jeśli aplikacja nie wymaga połączeń przychodzących poprzez serwer SOCKS, nie należy używać funkcji **Rbind()**.

Uwagi:

1. Klient FTP używa funkcji **Rbind()**, ponieważ protokół FTP pozwala serwerowi FTP na nawiązywanie połączeń danych na żądanie wysłania plików lub danych z klienta FTP.
2. Serwer SOCKS ustanawia połączenie dla danych z klientem FTP i przekazuje dane pomiędzy klientem a serwerem FTP. Wiele serwerów SOCKS zezwala na połączenia z chronionym klientem w ustalonym okresie czasu. Jeśli serwer nie połączy się w tym okresie, dla funkcji **accept()** wystąpi błąd ECONNABORTED.
3. Klient FTP inicjuje połączenie wychodzące TCP do mniej chronionej sieci poprzez serwer SOCKS. Adres docelowy podany przez klienta FTP podczas połączenia to adres IP i port serwera FTP znajdującego się w sieci niechronionej. System chronionego hosta został skonfigurowany na zakładce SOCKS do kierowania tego połączenia przez serwer SOCKS. Po skonfigurowaniu system automatycznie przekieruje połączenia do serwera SOCKS podanego na zakładce SOCKS.
4. Gniazdo jest otwierane i wywoływana jest funkcja **Rbind()** w celu nawiązania połączenia przychodzącego TCP. To połączenie pochodzi z tego samego adresu IP, który został podany powyżej. Połączenia przychodzące i wychodzące przez serwer SOCKS dla określonego wątku muszą być nawiązywane parami. Innymi słowy, wszystkie połączenia przychodzące **Rbind()** muszą być nawiązywane bezpośrednio po połączeniu wychodzącym przez serwer SOCKS, a przed uruchomieniem funkcji **Rbind()** nie mogą wystąpić żadne próby połączeń innych niż SOCKS, związanych z tym wątkiem.
5. Funkcja **getsockname()** zwraca adres serwera SOCKS. Gniazdo jest logicznie powiązane z adresem IP serwera SOCKS połączonym z portem wybranym przez serwer SOCKS. W przykładzie adres jest wysyłany przez ″połączenie kontrolne″ Socket CTLsd do serwera FTP umieszczonego w mniej chronionej sieci. Jest to adres, z którym łączy się serwer FTP. Serwer FTP łączy się z serwerem SOCKS, a nie bezpośrednio z chronionym serwerem.
6. Serwer SOCKS ustanawia połączenie dla danych z klientem FTP i przekazuje dane pomiędzy klientem a serwerem FTP. Wiele serwerów SOCKS zezwala na połączenia z chronionym klientem w ustalonym okresie czasu. Jeśli serwer nie połączy się w tym czasie, dla funkcji **accept()** wystąpi błąd ECONNABORTED.

Ochrona wątków

Funkcja realizuje ochronę wątków, jeśli można ją uruchomić jednocześnie w wielu wątkach w ramach tego samego procesu. Funkcja realizuje ochronę wątków wtedy i tylko wtedy, gdy wszystkie funkcje, które ta funkcja wywołuje, również realizują ochronę wątków. Funkcje API gniazd składają się z funkcji systemowych i sieciowych, które realizują ochronę wątków.

Wszystkie funkcje sieciowe, których nazwy kończą się przyrostkiem ″_r″, mają podobną semantykę i również realizują ochronę wątków. Przykładowy program wykorzystujący funkcję API gniazd z ochroną wątków znajduje się w sekcji Przykład: zastosowanie funkcji gethostbyaddr_r() w procedurach sieciowych z ochroną wątków.

Inne procedury translacji realizują wzajemnie ochronę wątków, ale korzystają ze struktury danych _res. Ta struktura danych jest współużytkowana pomiędzy wszystkimi wątkami w procesie i może być zmieniana przez aplikację podczas wywołania procedury translacji. Przykładowy program korzystający z procedur translacji znajduje się w sekcji Przykład: aktualizowanie i odpytywanie serwera DNS.

Nieblokujące operacje we/wy

Gdy aplikacja wywoła jedną z funkcji wejściowych gniazda, a nie będzie żadnych danych do odczytania, funkcja zablokuje się i nie zakończy się, dopóki nie zostaną przeczytane jakieś dane. Podobnie aplikacja może zablokować funkcję wyjściową gniazda, jeśli nie będzie mogła natychmiast wysłać danych. Funkcje **connect()** oraz **accept()** mogą również zablokować się podczas oczekiwania na ustanowienie połączenia z innymi programami.

Gniazda udostępniają aplikacjom metody umożliwiające wywoływanie funkcji, które się blokują w taki sposób, że mogą one być zakończone bez opóźnienia. Polegają one na wywołaniu funkcji **fcntl()**, która włącza opcję **O_NONBLOCK** lub na wywołaniu funkcji **ioctl()**, która włącza opcję **FIONBIO**. Po uruchomieniu trybu nieblokującego, jeśli funkcja nie może być zakończona bez zablokowania, zostaje ona zakończona natychmiast. Funkcja **connect()** może zwrócić parametr **[EINPROGRESS]**, który oznacza, że inicjowanie połączenia zostało uruchomione. Można później użyć funkcji **select()** do określenia, czy połączenie zostało nawiązane. Dla wszystkich funkcji, które mogą zostać uruchomione w trybie nieblokującym, kod błędu **[EWOULDBLOCK]** wskazuje, że wywołanie się nie powiodło.

Trybu nieblokującego można użyć z następującymi funkcjami gniazd:
- **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()**

Przykładowy program używający nieblokujących operacji we/wy znajduje się w sekcji Przykład: nieblokujące operacje we/wy i funkcja select().

Sygnały

Aplikacja może zażądać asynchronicznego powiadomienia (żądanie wysłania przez system **sygnału**) o wystąpieniu warunku, na który czeka. Są dwa sygnały asynchroniczne, które gniazda mogą przesłać do aplikacji.
1. **SIGURG** to sygnał wysyłany, gdy zostaną odebrane dane typu out-of-band (OOB) przez gniazdo obsługujące dane OOB. Na przykład gniazdo typu SOCK_STREAM z rodziną adresów AF_INET można skonfigurować, aby wysyłało sygnał SIGURG.
2. **SIGIO** to sygnał wysyłany, gdy w gnieździe dowolnego typu zostaną odebrane normalne dane, dane typu OOB, wystąpią warunki błędów lub zdarzy się cokolwiek innego.

Aplikacja powinna sprawdzić, czy jest w stanie obsłużyć otrzymane sygnały przed wysłaniem do systemu żądania przysyłania sygnałów. Można tego dokonać konfigurując **procedury obsługi sygnału**. Jednym ze sposobów ustawienia procedury obsługi sygnału jest wywołanie funkcji **sigaction()**.

Aplikacja wysyła do systemu żądanie wysłania sygnału **SIGURG** jedną z następujących metod:

- wywołując funkcję **fcntl()** i podając w parametrze komendy F_SETOWN identyfikator procesu lub grupy,
- wywołując funkcję **ioctl()** i określając komendę FIOSETOWN lub SIOCSPGRP (żądanie).

Aplikacja żąda od systemu wysyłania sygnału **SIGIO** w dwóch krokach. W pierwszym, jak opisano powyżej, należy ustawić dla sygnału **SIGURG** identyfikator procesu lub grupy, z jakim proces został uruchomiony. Dzięki temu system wie, gdzie ma być dostarczony sygnał. W drugim kroku aplikacja musi wykonać jedną z następujących czynności:

- wywołać funkcję **fcntl()** i podać komendę F_SETFL z opcją FASYNC,
- wywołać funkcję **ioctl()** i podać komendę FIOASYNC.

Zmusza to system do wygenerowania sygnału SIGIO. Należy zauważyć, że powyższe czynności można wykonać w dowolnej kolejności. Ponadto, jeśli aplikacja wysyła te żądania na gnieździe nasłuchującym, ustawiane wartości są dziedziczone przez wszystkie gniazda i zwracane do aplikacji przez funkcję **accept()**. Dzięki temu dopiero co zaakceptowane gniazdo będzie miało ten sam identyfikator procesu lub grupy, jak również te same informacje w odniesieniu do wysyłanego sygnału SIGIO.

Gniazdo może także generować sygnały asynchroniczne w przypadku wystąpienia warunków błędu. Zawsze, gdy aplikacja odbierze [EPIPE] jako wartość **errno** dla funkcji gniazda, do procesu, który wywołał operację zakończoną wartością **errno** wysyłany jest sygnał SIGPIPE. W implementacji BSD sygnał SIGPIPE domyślnie kończy proces, który zakończył się wartością **errno**. Aby zachować zgodność z poprzednimi wersjami systemu OS/400, implementacja OS/400 domyślnie ignoruje sygnał SIGPIPE.Dzięki temu dodanie funkcji sygnałów nie będzie miało ujemnego wpływu na istniejące aplikacje.

Dostarczenie sygnału do procesu, który jest zablokowany na funkcji gniazda, powoduje, że funkcja ta kończy oczekiwanie z wartością **[EINTR] errno**, umożliwiając działanie procedurze obsługi sygnału aplikacji. Dotyczy to funkcji:

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

Istotne znaczenie ma fakt, że sygnały nie udostępniają aplikacjom deskryptorów gniazd identyfikujących miejsce wystąpienia sygnalizowanego warunku. Jeśli aplikacja używa wielu deskryptorów gniazd, to musi albo odpytywać poszczególne deskryptory, albo wywołać funkcję **select()** w celu określenia przyczyny otrzymania sygnału.

Przykładowy program używający programów znajduje się w sekcji Przykład: zastosowanie sygnałów w blokujących funkcjach API.

Rozsyłanie grupowe IP

Rozsyłanie grupowe IP umożliwia aplikacjom wysłanie pojedynczego datagramu IP, który zostanie odebrany przez grupę hostów w sieci. Hosty należące do grupy mogą znajdować się w tej samej podsieci lub w różnych podsieciach połączonych routerami obsługującymi rozsyłanie grupowe. Hosty mogą być dołączane do grup i usuwane z nich w

każdej chwili. Nie ma ograniczeń dotyczących położenia ani liczby członków grupy hostów. Grupę hostów identyfikuje klasa D adresów internetowych w zakresie od 224.0.0.1 do 239.255.255.255.

Obecnie rozsyłania grupowego można używać tylko dla rodziny adresów AF_INET.

Aplikacja może wysyłać i odbierać datagramy rozsyłania grupowego za pomocą funkcji API gniazd i bezpołączeniowych gniazd typu SOCK_DGRAM. Rozsyłanie grupowe jest metodą transmisji typu jeden-do-wielu. Do rozsyłania grupowego nie można użyć zorientowanych na połączenie gniazd typu SOCK_STREAM. Po utworzeniu gniazda typu SOCK_DGRAM aplikacja może użyć funkcji **setsockopt()** do sterowania charakterystyką rozsyłania grupowego przypisaną do tego gniazda. Funkcja **setsockopt()** akceptuje następujące opcje poziomu IPPROTO_IP:

- IP_ADD_MEMBERSHIP: Dołącza do podanej grupy rozsyłania.
- IP_DROP_MEMBERSHIP: Opuszcza podaną grupę rozsyłania.
- IP_MULTICAST_IF: Konfiguruje interfejs, poprzez który wysyłane są wychodzące datagramy rozsyłania grupowego.
- IP_MULTICAST_TTL: Ustawia wartość Time To Live (TTL) w nagłówku IP wychodzących datagramów rozsyłania.
- IP_MULTICAST_LOOP: Określa, czy kopia wychodzącego datagramu rozsyłania ma być dostarczana do hosta wysyłającego, który jest członkiem grupy rozsyłania.

Przykłady zastosowań rozsyłania grupowego IP zawiera sekcja Przykład: korzystanie z rozsyłania grupowego.

Przesyłanie danych pliku – funkcje send_file() i accept_and_recv()

Obsługa gniazd w systemie OS/400 obejmuje funkcje API **send_file()** i **accept_and_recv()**, które umożliwiają szybsze i łatwiejsze przesyłanie plików przez połączone gniazda. Te dwie funkcje API są szczególnie użyteczne dla aplikacji przesyłających pliki, takich jak serwery HTTP.

Jedno wywołanie funkcji **send_file()** umożliwia wysłanie danych pliku bezpośrednio z systemu plików poprzez połączone gniazdo.

Funkcja **accept_and_recv()** jest kombinacją trzech funkcji gniazd: **accept()**, **getsockname()** i **recv()**.

Przykładowe programy dla funkcji **send_file()** i **accept_and_recv()** można znaleźć w sekcji Przykład: przesyłanie danych pliku za pomocą funkcji send_file() i accept_and_recv().

Dane spoza pasma

Dane spoza pasma (Out-of-band - OOB) są danymi specyficznymi dla użytkownika, które mają znaczenie tylko dla gniazd zorientowanych na połączenie (strumieniowych). Dane strumieniowe są zwykle odbierane w takiej samej kolejności, w jakiej zostały wysłane. Dane OOB są otrzymywane niezależnie od ich pozycji w strumieniu (niezależnie od kolejności, w jakiej zostały wysłane). Dzieje się tak, ponieważ podczas przesyłania tak oznaczonych danych z programu A do programu B, program B jest powiadamiany o ich przyjściu.

Dane OOB (out-of-band) są obsługiwane tylko przez rodziny adresów AF_INET (SOCK_STREAM) i AF_INET6 (SOCK_STREAM).

Dane te można wysyłać określając opcję MSG_OOB w funkcjach **send()**, **sendto()** i **sendmsg()**.

Transmisja danych OOB jest taka sama, jak zwyczajnych danych. Są one wysyłane po danych buforowanych. Inaczej mówiąc, dane OOB nie mają pierwszeństwa przed buforowanymi danymi; są przesyłane w takiej kolejności, w jakiej zostały wysłane.

Po stronie odbierającej jest to bardziej skomplikowane:

- Funkcja API gniazd śledzi dane OOB, które są otrzymywane przez system, za pomocą znacznika OOB. Znacznik ten wskazuje na ostatni bajt w danych OOB, które zostały wysłane.

Uwaga: Wartość wskazująca, który bajt wskazuje znacznik OOB, jest ustawiana globalnie dla całego systemu (wszystkie aplikacje korzystają z tej wartości). Wartość ta musi być spójna pomiędzy lokalnym i zdalnym końcem połączenia TCP. Musi być używana zgodnie pomiędzy aplikacjami typu klient i typu serwer. Sposób zmiany bajtu wskazującego znacznik OOB opisano w sekcji Komenda Zmiana atrybutów TCP/IP (Change TCP Attributes - CHGTCPA) w Centrum informacyjnym.

Żądanie **SIOCATMARK ioctl()** określa, czy wskaźnik odczytu pokazuje ostatni bajt OOB.

Uwaga: Jeśli wysłano wiele bloków danych OOB, znacznik OOB będzie wskazywał ostatni bajt końcowego wystąpienia danych OOB.

- Jeśli zostały wysłane dane OOB, operacja wejściowa będzie przetwarzała dane do znacznika OOB, niezależnie od tego, czy dane OOB zostały odebrane.
- Do odbierania danych OOB używane są funkcje **recv()**, **recvmsg()** lub **recvfrom()** (z ustawioną opcją MSG_OOB). Jeśli zdarzy się jeden z wymienionych przypadków, zwracany jest błąd [EINVAL]:
  - opcja gniazda SO_OOBINLINE nie została ustawiona i nie ma danych OOB do odebrania,
  - została ustawiona opcja gniazda SO_OOBINLINE.

  Jeśli opcja gniazda SO_OOBINLINE nie została ustawiona, a program wysyłający wysłał dane OOB o wielkości przekraczającej jeden bajt, wszystkie bajty, oprócz ostatniego, są uznawane za normalne dane. (Zwykłe dane to takie, które program może odebrać bez określania opcji MSG_OOB). Ostatni bajt z wysłanych danych OOB nie jest umieszczany w normalnym strumieniu danych. Można go odtworzyć tylko poprzez wywołanie funkcji **recv()**, **recvmsg()** lub **recvfrom()** z ustawioną opcją MSG_OOB. Jeśli funkcja odbierająca jest wywołana bez ustawionej opcji MSG_OOB i odbierane są normalne dane, bajt OOB jest usuwany. Ponadto jeśli wysłano wiele danych OOB, wcześniejsze dane są niszczone, a zapamiętywana jest tylko ostatnia pozycja danych OOB.

  Jeśli zostanie ustawiona opcja gniazda SO_OOBINLINE, wtedy wszystkie wysyłane dane są przechowywane w normalnym strumieniu danych. Dane można odtworzyć przez wywołanie jednej z trzech funkcji odbierających, bez określania opcji MSG_OOB (jeśli zostanie ona określona, zostanie zwrócony błąd [EINVAL]). Jeśli wysłanych zostanie wiele wystąpień danych OOB, to nie zostaną one utracone.
- Dane OOB nie są niszczone, jeśli nie zostanie ustawiona opcja SO_OOBINLINE; zostają one odebrane, a następnie użytkownik włącza opcję SO_OOBINLINE. Początkowy bajt OOB jest uważany za normalne dane.
- Jeśli opcja SO_OOBINLINE nie została ustawiona, dane OOB zostały wysłane, a program odbierający wywołał funkcję wejścia do odbioru danych OOB, wtedy znacznik będzie w dalszym ciągu poprawny. Program odbierający może w dalszym ciągu sprawdzić, czy odczytany wskaźnik wskazuje znacznik OOB, nawet jeśli bajt OOB został odczytany.

Multipleksowanie operacji we/wy – funkcja select()

Asynchroniczne operacje we/wy zapewniają znacznie efektywniejszą metodę maksymalizowania zasobów aplikacji niż funkcja API **select()**, dlatego zalecane jest używanie tych operacji. Jednak konkretny projekt aplikacji może dopuszczać użycie funkcji **select()**. Podobnie jak asynchroniczne operacje we/wy funkcja **select()** tworzy wspólny punkt jednoczesnego oczekiwania dla wielu warunków. Jednak funkcja **select()** umożliwia aplikacji określenie zestawu deskryptorów w celu:

- sprawdzenia, czy są dane do odczytu,
- sprawdzenia, czy są dane do zapisania,
- sprawdzenia, czy wystąpił wyjątek.

Deskryptory, które można określić w każdym zestawie, mogą być deskryptorami gniazd, deskryptorami plików lub deskryptorami dowolnych innych obiektów, które mogą być reprezentowane przez deskryptory.

Funkcja **select()** umożliwia również aplikacjom określenie, czy chcą pozostać niedostępne w czasie oczekiwania na dane. Aplikacje mogą określić długość oczekiwania. Przykładowy program znajduje się w sekcji Przykład: nieblokujące operacje we/wy i funkcja select().

Funkcje sieciowe gniazd

Funkcje sieciowe gniazd umożliwiają aplikacjom uzyskiwanie informacji od hostów, protokołów, usług oraz plików sieciowych. Dostęp do tych informacji możliwy jest poprzez nazwę, adres lub poprzez dostęp sekwencyjny. Te funkcje sieciowe (lub procedury) są wymagane podczas konfigurowania komunikacji pomiędzy programami, które działają w sieci, i nie są używane przez gniazda rodziny AF_UNIX. Krótkie podsumowanie poszczególnych funkcji sieciowych zawiera sekcja Funkcje (procedury) sieciowe gniazd w skorowidzu funkcji API w Centrum informacyjnym.

Procedury wykonują następujące zadania:

- odwzorowują nazwy hostów na adresy sieciowe,
- odwzorowują nazwy sieciowe na numery sieci,
- odwzorowują nazwy protokołów na numery protokołów,
- odwzorowują nazwy usług na numery portów,
- przekształcają kolejność bajtów w adresach sieciowych Internetu,
- przekształcają adresy internetowe i notacje w postaci dziesiętnej.

W skład procedur sieciowych wchodzą tzw. procedury tłumaczące. Służą one do tworzenia, wysyłania i interpretowania pakietów dla serwerów nazw w domenie internetowej oraz do tłumaczenia nazw. Procedury tłumaczące są zwykle wywoływane przez funkcje **gethostbyname()** , **gethostbyaddr()**, **getnameinfo()** i **getaddrinfo()**, ale mogą być również wywoływane bezpośrednio. Przykłady programów korzystających z procedur tłumaczących znajdują się w sekcji Przykład: zastosowanie funkcji gethostbyaddr_r() w procedurach sieciowych z ochroną wątków. Procedury tłumaczące są używane głównie przez aplikacje używające gniazd do dostępu do systemu nazw domen (DNS). Szczegółowe informacje o tym, jak można używać gniazd z systemem DNS przedstawiono w sekcji Obsługa systemu nazw domen (DNS).

Obsługa systemu nazw domen (DNS)

Serwer iSeries zapewnia aplikacjom dostęp do systemu nazw domen (DNS) poprzez funkcje tłumaczące. Na system DNS składają się trzy główne komponenty:

- **Przestrzeń nazw domen i rekordy zasobów**
  Określają one przestrzeń nazw o strukturze drzewa oraz dane związane z nazwami.
- **Serwery nazw**
  Są to programy serwera, które przechowują informacje o strukturze drzewa domeny. Więcej informacji o serwerach nazw można znaleźć w Centrum informacyjnym, w temacie DNS.
- **Programy tłumaczące**
  Są to programy, które w odpowiedzi na żądania klientów wyodrębniają informacje z serwerów nazw.

Programy tłumaczące dostępne w implementacji systemu OS/400 są funkcjami gniazd pozwalającymi na komunikację z serwerem nazw. Procedury te są używane do tworzenia, wysyłania, aktualizowania i interpretowania pakietów oraz do przechowywania nazw w pamięci podręcznej w celu zwiększenia wydajności. Udostępniają one również funkcje konwersji z ASCII na EBCDIC i z EBCDIC na ASCII. Opcjonalnie, do bezpiecznej komunikacji z serwerem DNS programy tłumaczące używają sygnatur transakcyjnych (TSIG). Krótkie podsumowanie poszczególnych procedur tłumaczących zawiera sekcja Funkcje (procedury) sieciowe gniazd w skorowidzu funkcji API w Centrum informacyjnym. Tam również znajdują się informacje o strukturze danych _res. Struktura _res zawiera globalne informacje używane przez procedury tłumaczące.

Więcej informacji o nazwach domen zawierają następujące dokumenty RFC, które można znaleźć, posługując się wyszukiwarką RFC.

- RFC 1034, ″Domain names - concepts and facilities″ (Nazwy domen - koncepcje i narzędzia)
- RFC 1035, ″Domain names - implementation and specification″ (Nazwy domen - implementacja i specyfikacja)
- RFC 1886, ″DNS Extensions to support IP version 6″ (Rozszerzenia DNS obsługujące protokół IP wersja 6)
- RFC 2136, ″Dynamic Updates in the Domain Name System (DNS UPDATE)″ (Dynamiczne aktualizacje systemu nazw domen - DNS UPDATE)

- RFC 2181, ″Clarifications to the DNS Specification″ (Objaśnienia specyfikacji DNS)
- RFC 2845, ″Secret Key Transaction Authentication for DNS (TSIG)″ (Uwierzytelnianie DNS za pomocą klucza tajnego - TSIG)
- RFC 3152, ″DNS Delegation of IP6.ARPA″ (Przeniesienie DNS do adresów IP6.ARPA)

Więcej informacji dotyczących innych sposobów użycia systemu DNS przez aplikacje gniazd zawierają następujące tematy:

- Zmienne środowiskowe
  Temat ten opisuje zmienne środowiskowe, których można użyć do rozstrzygania nazw.
- Buforowanie danych
  Temat ten zawiera szczegółowe informacje dotyczące użycia gniazd do buforowania odpowiedzi na zapytania serwera DNS w celu zmniejszenia ruchu w sieci. Sekcja Przykład: aktualizowanie i odpytywanie serwera DNS zawiera program przykładowy ilustrujący, jak można odpytywać i aktualizować rekordy DNS za pomocą funkcji API gniazd.

Zmienne środowiskowe

Zmiennych środowiskowych można używać do wymuszania domyślnych parametrów początkowych funkcji tłumaczenia nazw. Zmienne środowiskowe są sprawdzane tylko po zakończonym powodzeniem wywołaniu funkcji **res_init()** lub **res_ninit()**. Zatem, jeśli struktura została zainicjowana ręcznie, zmienne środowiskowe są ignorowane. Należy także zwrócić uwagę, iż struktura jest inicjowana tylko raz, dlatego późniejsze modyfikacje zmiennych środowiskowych również będą ignorowane.

Uwaga: Nazwa zmiennej środowiskowej musi byś pisana wielkimi literami. Wartości łańcuchów mogą zawierać zarówno małe, jak i wielkie litery. W systemach japońskich używających identyfikatora CCSID 290 należy zarówno w nazwach zmiennych środowiskowych, jak i wartościach, używać wyłącznie cyfr i wielkich liter. Poniższa lista zawiera opisy zmiennych środowiskowych, których można używać razem z funkcjami API **res_init()** i **res_ninit()**.

LOCALDOMAIN
Tej zmiennej środowiskowej można przypisać oddzieloną odstępami listę maksymalnie sześciu domen do wyszukiwania. Zmienna może mieć maksymalnie 256 znaków (razem z odstępami). Spowoduje to przesłonięcie skonfigurowanej listy wyszukiwania (struktury state.defdname i state.dnsrch). Jeśli zostanie określona lista wyszukiwania, w zapytaniach nie będzie używana domyślna domena lokalna.

RES_OPTIONS
Umożliwia modyfikację pewnych wewnętrznych zmiennych programu tłumaczącego. Zmienna ta może przyjmować jedną lub więcej opisanych niżej opcji, oddzielonych odstępami.

- **NDOTS:n**
  Określa maksymalną liczbę kropek, jakie może zawierać nazwa przekazana do funkcji **res_query()**, zanim zostanie wykonane zapytanie absolutne. Wartością domyślną jest ″1″, tzn. jeśli w nazwie występuje przynajmniej jedna kropka, najpierw zostanie podjęta próba przetłumaczenia tej nazwy jako absolutnej, a w razie niepowodzenia będą do niej po kolei dołączane poszczególne elementy listy wyszukiwania.
- **TIMEOUT:n**
  Określa czas (w sekundach), przez jaki program tłumaczący będzie oczekiwał na odpowiedź ze zdalnego serwera nazw, zanim powtórzy zapytanie.
- **ATTEMPTS:n**
  Określa liczbę zapytań, jakie program tłumaczący będzie wysyłał do serwera nazw, zanim zacznie odpytywać następny serwer na liście.
- **ROTATE**
  Ustawia wartość RES_ROTATE w strukturze danych _res.options, która powoduje rotację serwerów nazw na liście. Powoduje to równomierne rozłożenie obciążenia związanego z zapytaniami pomiędzy serwery, co zapobiega sytuacji, gdy wszyscy klienci za każdym razem zaczynają odpytywanie od pierwszego serwera na liście.
- **NO-CHECK-NAMES**
  Ustawia wartość RES_NOCHECKNAME w strukturze danych _res.options, która wyłącza używaną w

nowszych wersjach serwera BIND funkcję sprawdzania nazw hostów i adresów pocztowych pod kątem niepoprawnych znaków, takich jak podkreślenie (_), znaki spoza zestawu ASCII i znaki kontrolne.

QIBM_BIND_RESOLVER_FLAGS
Ta zmienna środowiskowa zawiera oddzieloną odstępami listę opcji programu tłumaczącego. Spowoduje to przesłonięcie opcji RES_DEFAULT (struktura state.options) i wartości skonfigurowanych w systemie komendą Zmiana domeny TCP/IP (Change TCP/IP Domain - CHGTCPDMN). Struktura state.options zostanie zainicjowana w zwykły sposób, za pomocą opcji RES_DEFAULT, wartości środowiskowych OPTIONS oraz wartości konfiguracyjnych komendy CHGTCPDMN. Następnie zostanie ona użyta do przesłonięcia tych wartości domyślnych. Określone w tej zmiennej środowiskowej opcji można poprzedzić symbolami '+', '-' lub 'NOT_', aby ustawić ('+') lub zresetować ('-','NOT_') wartość.

Na przykład, aby ustawić opcję RES_NOCHECKNAME i wyłączyć RES_ROTATE, z interfejsu znakowego należy użyć następującej komendy:

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

lub

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

QIBM_BIND_RESOLVER_SORTLIST
Ta zmienna środowiskowa zawiera oddzieloną odstępami listę maksymalnie dziesięciu par adresów IP/masek w postaci oddzielonych kropkami liczb dziesiętnych (9.5.9.0/255.255.255.0), stanowiącą listę sortowania (struktura state.sort_list).

Buforowanie danych

Buforowanie odpowiedzi na zapytania do serwera DNS jest wykonywane przez gniazda systemu OS/400 w celu zmniejszenia ruchu w sieci. Pamięć podręczna jest dodawana i aktualizowana, gdy jest to wymagane.

Jeśli w _res.options zostanie ustawiona opcja RES_AAONLY (tylko autorytatywne odpowiedzi), zapytania będą zawsze wysyłane do sieci. W takim przypadku system nigdy nie sprawdza, czy odpowiedzi są w pamięci podręcznej. Jeśli opcja RES_AAONLY nie jest ustawiona, przed wysłaniem zapytania do sieci system szuka odpowiedzi w pamięci podręcznej . Jeśli system znajdzie odpowiedź, której czas życia nie upłynął, zwraca ją do użytkownika jako odpowiedź na zapytanie. W przypadku, gdy czas życia odpowiedzi upłynął, pozycja ta jest usuwana, a system wysyła zapytanie do sieci. Również wtedy, kiedy odpowiedzi nie ma w pamięci podręcznej, zapytanie zostanie wysłane do sieci.

Jeśli odpowiedzi z sieci są autorytatywne, są one zapisywane w pamięci podręcznej. Odpowiedzi nieautorytatywne nie są buforowane. Także odpowiedzi odebrane jako wynik zapytania odwrotnego nie są buforowane. Pamięć podręczną można skasować aktualizując konfigurację serwera DNS za pomocą komendy CHGTCPDMN, opcji 12 komendy CFGTCP lub przy użyciu programu iSeries Navigator.

Przykładowy program korzystający z buforowania danych znajduje się w sekcji Przykład: aktualizowanie i odpytywanie serwera DNS.

Zgodność z Berkeley Software Distributions (BSD)

Gniazda są interfejsem systemu Berkeley Software Distributions (BSD). Semantyka, taka jak kody powrotu otrzymywane przez aplikację i argumenty dostępne w obsługiwanych funkcjach, należy do systemu BSD. Niektóre reguły semantyki BSD nie są dostępne w implementacji OS/400 i aby uruchomić w systemie typową aplikację opartą na gniazdach BSD, mogą być potrzebne zmiany.

Poniższa lista podsumowuje różnice pomiędzy implementacjami OS/400 i BSD:

/etc/hosts, /etc/services, /etc/networks i /etc/protocols
Dla tych plików implementacja OS/400 dostarcza następujące zbiory bazy danych, które obsługują odpowiednio te same funkcje.

Zbiór QUSRSYS	Zawartość

QATOCHOST	Lista nazw hostów i związanych z nimi adresów IP.
QATOCPN	Lista sieci i związanych z nimi adresów IP.
QATOCPP	Lista protokołów używanych w Internecie.
QATOCPS	Lista usług oraz portów i protokołów używanych przez te usługi.

/etc/resolv.conf
Implementacja OS/400 wymaga, aby te informacje zostały skonfigurowane za pomocą zakładki właściwości TCP/IP w programie iSeries Navigator.W celu użycia zakładki właściwości TCP/IP, wykonaj następujące czynności:

1. W programie iSeries Navigator rozwiń **Serwer iSeries --> Sieć --> Konfiguracja TCP/IP**.
2. Kliknij prawym klawiszem myszy **Konfiguracja TCP/IP**.
3. Kliknij **Właściwości**.

bind() W systemie BSD klient może utworzyć gniazdo AF_UNIX za pomocą funkcji socket(), połączyć się z serwerem za pomocą funkcji connect(), a następnie powiązać nazwę z gniazdem za pomocą funkcji bind(). Implementacja OS/400 nie obsługuje tego scenariusza (funkcja bind() nie powiedzie się).

close()

Implementacja OS/400 obsługuje licznik czasu zwłoki dla funkcji close(), z wyjątkiem gniazd AF_INET w sieciach SNA. Niektóre implementacje BSD nie obsługują licznika czasu zwłoki dla funkcji close().

connect()
Wywołanie w systemie BSD funkcji connect() w odniesieniu do gniazda, które zostało już połączone z adresem i korzysta z usługi bezpołączeniowego transportu z użyciem niepoprawnego adresu lub adresu o niepoprawnej długości, powoduje rozłączenie gniazda. Implementacja OS/400 nie obsługuje tego scenariusza (funkcja connect() nie powiedzie się i gniazdo pozostanie połączone).

Gniazdo z transportem bezpołączeniowym, dla którego została wywołana funkcja connect(), można odłączyć, przypisując parametrowi address_length wartość zero i wywołując kolejną funkcję connect().

accept(), getsockname(), getpeername(), recvfrom() i recvmsg()
W przypadku rodziny adresów AF_UNIX lub AF_UNIX_CCSID, gdy gniazdo nie zostanie znalezione, domyślna implementacja OS/400 może zwrócić długość adresu równą zero i nieokreśloną strukturę adresu. Implementacje OS/400 BSD 4.4/UNIX 98 i inne mogą zwrócić małą strukturę adresu zawierającą tylko określenie rodziny adresów.

ioctl()

- W systemie BSD dla gniazd typu SOCK_DGRAM żądanie FIONREAD zwróci długość danych plus długość adresu. W implementacji OS/400 żądanie FIONREAD zwraca tylko długość danych.
- Nie wszystkie żądania dostępne w większości implementacji BSD funkcji **ioctl()** są dostępne w implementacji OS/400 funkcji **ioctl()**.

listen() W systemie BSD wywołanie funkcji **listen()** z parametrem backlog ustawionym na wartość mniejszą niż zero nie spowoduje błędu. Dodatkowo, w niektórych przypadkach implementacja BSD nie korzysta z parametru backlog lub używa algorytmu do obliczenia końcowej wartości tego parametru. Jeśli wartość parametru backlog jest mniejsza od zera, implementacja OS/400 zwraca błąd. Nadanie parametrowi backlog prawidłowej wartości powoduje, że zostanie ona użyta jako ten parametr. Jednakże nadanie parametrowi backlog wartości większej niż {SOMAXCONN} spowoduje, że parametr przyjmie wartość domyślną ustawioną dla {SOMAXCONN}.

Dane Out-of-band (OOB)
W implementacji OS/400 dane OOB nie są niszczone, jeśli opcja SO_OOBINLINE nie została ustawiona, dane OOB zostały odebrane i użytkownik włączył opcję SO_OOBINLINE. Początkowy bajt OOB jest uważany za normalne dane.

parametr protokołu funkcji socket()

Dodatkowym zabezpieczeniem jest zablokowanie wszystkim użytkownikom możliwości utworzenia gniazda SOCK_RAW z podaniem protokołu IPPROTO_TCP lub IPPROTO_UDP.

res_xlate() i res_close()

W implementacji OS/400 funkcje te są wbudowane w procedury tłumaczące. Funkcja **res_xlate()** tłumaczy pakiety DNS z EBCDIC na ASCII i z ASCII na EBCDIC. Funkcja **res_close()** jest używana do zamknięcia gniazda, które zostało użyte przez funkcję **res_send()** z ustawioną opcją RES_STAYOPEN. Powoduje to także zresetowanie struktury _res.

sendmsg() i recvmsg()

Implementacja OS/400 funkcji **sendmsg()** i **recvmsg()** pozwala na użycie do {MSG_MAXIOVLEN} (włącznie) wektorów we/wy. Implementacja BSD dopuszcza maksymalnie {MSG_MAXIOVLEN - 1} wektorów we/wy.

shutdown()

W implementacji OS/400 funkcja wyjściowa, która jest w danej chwili zablokowana dla deskryptora gniazda, pozostanie zablokowana po wywołaniu funkcji **shutdown()**. W implementacji BSD blokująca funkcja wyjściowa jest zakończona z wartością kodu błędu [EPIPE]. Podobnie implementacja BSD kończy blokujące operacje wejściowe z wyjściową wartością zerową, gdy są one blokujące i inny proces lub wątek wywoła funkcję **shutdown()**. W implementacji OS/400 taka sytuacja powoduje, że wszystkie kolejne funkcje wejściowe z zerową wartością wyjściową nie powiodą się, ale blokująca funkcja wejściowa kontynuuje blokadę, dopóki dane nie zostaną odebrane lub nie zostaną podjęte inne czynności w celu wyprowadzenia jej ze stanu oczekiwania.

Sygnały

Istnieje kilka różnic związanych z obsługą sygnałów:

- Implementacje BSD wywołują sygnał SIGIO za każdym razem, gdy otrzymywane jest potwierdzenie dla danych wysyłanych przez operację wyjściową. Implementacja gniazd OS/400 nie generuje sygnałów powiązanych z danymi wychodzącymi.
- W implementacjach BSD działaniem domyślnym dla sygnału SIGPIPE jest zakończenie procesu. W celu zachowania kompatybilności z poprzednimi wersjami OS/400 implementacja OS/400 domyślnie ignoruje sygnał SIGPIPE.

opcja SO_REUSEADDR

W systemach BSD wywołanie funkcji **connect()** dla gniazda z rodziny AF_INET i rodzaju SOCK_DGRAM spowoduje, że system zmieni adres, z którym gniazdo jest powiązane, na adres interfejsu używanego do osiągnięcia adresu podanego w funkcji **connect()**. Jeśli na przykład gniazdo typu SOCK_DGRAM zostanie powiązane z adresem INADDR_ANY, a następnie połączone z adresem a.b.c.d, system zmieni gniazdo tak, że będzie ono powiązane z adresem IP interfejsu wybranego do kierowania pakietów do adresu a.b.c.d. Ponadto, jeśli adres IP, z którym powiązane jest gniazdo, to na przykład a.b.c.e, adres a.b.c.e pojawi się podczas wywołania funkcji **getsockname()** zamiast adresu INADDR_ANY i trzeba będzie użyć opcji SO_REUSEADDR, aby powiązać wszelkie inne gniazda dla tego samego numeru portu z adresem a.b.c.e.

Implementacja OS/400 zachowa się w tym przypadku inaczej, to jest nie zmieni lokalnego adresu z INADDR_ANY na a.b.c.e. Funkcja **getsockname()** będzie zwracać adres INADDR_ANY także po zrealizowaniu połączenia.

opcje SO_SNDBUF i SO_RCVBUF

Wartości nadawane opcjom SO_SNDBUF i SO_RCVBUF w systemie BSD zapewniają większą kontrolę niż w implementacji OS/400. W implementacji OS/400 wartości te mają znaczenie pomocnicze.

Zgodność z UNIX 98

Opracowany przez Open Group, konsorcjum programistów i dostawców oprogramowania, standard UNIX 98 polepszył współdziałanie systemów operacyjnych UNIX, definiując większość funkcji związanych z obsługą Internetu, z których system UNIX stał się znany. Od wersji V5R2 gniazda systemu OS/400 umożliwiają programistom pisanie kompatybilnych ze środowiskiem operacyjnym UNIX 98 aplikacji gniazd.Obecnie IBM obsługuje dwie wersje

większości funkcji API gniazd. Podstawowe funkcje API systemu OS/400 obsługują struktury i składnię BSD 4.3. Druga wersja gniazd używa składni i struktur zgodnych z 4.4 i specyfikacją interfejsu programistycznego UNIX 98. Definiując wartość 520 lub większą dla makra _XOPEN_SOURCE, można wybrać interfejs zgodny ze standardem UNIX 98.

Różnice w strukturze adresów dla aplikacji zgodnych zUNIX 98

Po podaniu makra _XOPEN_OPEN można pisać aplikacje zgodne z UNIX 98, używające tych samych rodzin adresów, które są używane w domyślnych implementacjach systemu OS/400; jednakże w strukturze adresów **sockaddr** występują różnice. W poniższej tabeli porównano strukturę adresów **sockaddr** BSD 4.3 ze strukturą adresów zgodną z UNIX 98:

Tabela 17. Porównanie struktur adresowych gniazd BSD 4.3 i UNIX 98/BSD 4.4

Struktura BSD 4.3	Struktura zgodna z BSD 4.4/UNIX 98
Struktura adresów sockaddr	

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];
	};

Struktura adresów sockaddr_in	

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];
	};

Struktura adresów sockaddr_in6	

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;
	};

Struktura adresów sockaddr_un	

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]
	};

Różnice w funkcjach API

Podczas kompilacji aplikacji w językach opartych na środowisku ILE z podaniem makra _XOPEN_SOURCE niektóre funkcje API gniazd są odwzorowywane na nazwy wewnętrzne. Nazwy te działają tak samo jak oryginalne funkcje API. Poniższa tabela zawiera funkcje API, których to dotyczy. Pisząc aplikację gniazd w języku opartym na C można określać wewnętrzne nazwy tych funkcji API bezpośrednio. Aby przeczytać uwagi dotyczące użycia i szczegóły związane z obiema wersjami tych funkcji API, należy użyć odsyłaczy.

Tabela 18. Funkcja API i odpowiadająca jej nazwa zgodna z UNIX 98

Nazwa funkcji API	Nazwa wewnętrzna
accept()	qso_accept98()

Tabela 18. Funkcja API i odpowiadająca jej nazwa zgodna z UNIX 98 (kontynuacja)

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()
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()

Tabela 18. Funkcja API i odpowiadająca jej nazwa zgodna z UNIX 98 (kontynuacja)

setnetent()	qso_setnetent98()
setprotoent()	qso_setprotoent98()
setservent()	qso_setprotoent98()
setsockopt()	qso_setsockopt98()
shutdown()	qso_shutdown98()
socket()	qso_socket98()
socketpair()	qso_socketpair98()

Przekazywanie deskryptorów pomiędzy procesami – funkcje sendmsg() i recvmsg()

Możliwość przekazania otwartego deskryptora pomiędzy zadaniami może prowadzić do nowego sposobu projektowania aplikacji klient/serwer. Przekazanie otwartego deskryptora między zadaniami pozwala jednemu procesowi, zwykle procesowi serwera, zrobić wszystko, co jest wymagane do uzyskania deskryptora (otworzyć plik, nawiązać połączenie, czekać na funkcję API **accept()** w celu zakończenia), a także pozwala innemu procesowi, zwykle procesowi roboczemu, obsłużyć wszystkie operacje przesyłania danych, kiedy deskryptor jest otwarty. Upraszcza to logikę zadań serwera i procesu roboczego. Konstrukcja taka umożliwia także łatwą obsługę różnych typów procesów roboczych. Serwer może łatwo sprawdzić, który rodzaj procesu roboczego powinien otrzymać deskryptor.

Gniazda udostępniają trzy zestawy funkcji API, które mogą przekazywać deskryptory pomiędzy zadaniami serwera:

- **spawn()**

  Uwaga: Funkcja **spawn()** nie jest funkcją API gniazd. Jest ona dostarczana jako część funkcji API systemu OS/400 związanych z procesami.
- **givedescriptor()** i **takedescriptor()**
- **sendmsg()** i **recvmsg()**

Funkcja API **spawn()** uruchamia nowe zadanie serwera (często zwane ″zadaniem potomnym″) i daje mu pewne deskryptory. Jeśli zadanie potomne jest już aktywne, należy użyć funkcji API **givedescriptor()** i **takedescriptor()** lub **sendmsg()** i **recvmsg()**.

Jednak funkcje API **sendmsg()** i **recvmsg()** mają więcej zalet w porównaniu z funkcjami **spawn()**, **givedescriptor()** i **takedescriptor()**:

Przenośność

Funkcje **givedescriptor()** i **takedescriptor()** są niestandardowe i unikalne dla iSeries. Jeśli ważna jest przenośność aplikacji między serwerem iSeries a systemem UNIX, lepiej będzie użyć funkcji API **sendmsg()** i **recvmsg()**.

Komunikacja informacji sterujących

Często, gdy zadanie procesu roboczego otrzyma deskryptor, potrzebuje ono dodatkowych informacji:

- jaki to rodzaj deskryptora,
- co zadanie procesu roboczego powinno z nim zrobić.

Funkcje **sendmsg()** i **recvmsg()** umożliwiają przesyłanie danych, które mogą być informacjami sterującymi, wraz z deskryptorem. Funkcje **givedescriptor()** i **takedescriptor()** nie mają takiej możliwości.

Wydajność

Aplikacje korzystające z funkcji **sendmsg()** i **recvmsg()** mają lepszą wydajność niż te, które korzystają z funkcji **givedescriptor()** i **takedescriptor()**, pod trzema względami:

- czasu, który upłynął,
- wykorzystania jednostki centralnej,

- skalowalności.

Wzrost wydajności aplikacji zależy od zakresu, w jakim aplikacja przekazuje deskryptory.

Pula zadań procesów roboczych
Można ustawić pulę zadań procesów roboczych tak, aby serwer mógł przekazać deskryptor i aby tylko jedno zadanie z puli go otrzymało. Aby to osiągnąć, należy użyć funkcji **sendmsg()** i **recvmsg()**, które spowodują, że wszystkie procesy robocze będą oczekiwały na współużytkowany deskryptor. Gdy serwer wywoła funkcję **sendmsg()**, tylko jedno z zadań procesu roboczego otrzyma deskryptor.

Nieznany ID procesu roboczego
Funkcja **givedescriptor()** wymaga, aby zadanie serwera znało ID zadania procesu roboczego. Zazwyczaj zadanie procesu roboczego uzyskuje ID zadania i przekazuje go do zadania serwera z kolejką danych. Funkcje **sendmsg()** i **recvmsg()** nie wymagają dodatkowych czynności przy tworzeniu i zarządzaniu tą kolejką danych.

Adaptacyjny program serwera
Gdy serwer jest zaprojektowany za pomocą funkcji **givedescriptor()** i **takedescriptor()**, zazwyczaj korzysta się z kolejki danych, aby przekazać identyfikatory zadań od zadań roboczych do serwera. Następnie serwer wykonuje **socket()**, **bind()**, **listen()** i **accept()**. Gdy zakończy się funkcja **accept()**, serwer przewinie do następnego dostępnego ID zadania z kolejki danych. Następnie przekaże połączenie przychodzące do zadania procesu roboczego. Problemy pojawiają się, gdy wystąpi jednocześnie wiele żądań połączeń przychodzących, dla których nie ma wystarczającej liczby zadań procesu roboczego. Jeśli kolejka danych zawierająca identyfikatory zadania procesu roboczego jest pusta, serwer blokuje się w oczekiwaniu na dostępność zadania roboczego lub tworzy dodatkowe zadania robocze. W wielu środowiskach nie jest wskazana żadna z tych możliwości, gdyż dodatkowe żądania przychodzące mogą wypełnić nasłuchujący backlog.

Serwery, które korzystają z funkcji **sendmsg()** i **recvmsg()** do przekazywania deskryptorów, nie odczuwają skutków dużego obciążenia, ponieważ nie muszą one wiedzieć, który proces roboczy będzie obsługiwał każde przychodzące połączenie. Gdy serwer wywoła **sendmsg()**, deskryptor połączenia przychodzącego i inne dane sterujące zostają umieszczone w wewnętrznej kolejce dla gniazda AF_UNIX. Gdy zadanie procesu roboczego stanie się dostępne, wywoła **recvmsg()** i otrzyma pierwszy deskryptor i dane sterujące z kolejki.

Nieaktywne zadanie procesu roboczego

Funkcja **givedescriptor()** wymaga, aby zadanie procesu roboczego było aktywne, funkcja **sendmsg()** nie wymaga tego. Zadanie wywołujące **sendmsg()** nie wymaga żadnych informacji o zadaniu procesu roboczego. Funkcja **sendmsg()** wymaga tylko ustanowienia połączenia z gniazdem AF_UNIX.

Oto przykład, jak można użyć funkcji **sendmsg()** do przekazania deskryptora do zadania, które nie istnieje:

Serwer może użyć funkcji **socketpair()** do utworzenia pary gniazd AF_UNIX, funkcji **sendmsg()** do wysłania deskryptora przez jedno z gniazd AF_UNIX utworzonych przez funkcję **socketpair()**, a następnie wywołać funkcję **spawn()**, aby utworzyć zadanie potomne, które odziedziczy drugie zakończenie pary gniazd. Zadanie potomne wywołuje **recvmsg()** w celu otrzymania deskryptora przekazanego przez serwer. Zadanie potomne nie było aktywne, gdy serwer wywoływał **sendmsg()**.

Jednoczesne przekazanie kilku deskryptorów
Funkcje **givedescriptor()** i **takedescriptor()** umożliwiają przekazywanie deskryptorów pojedynczo. Funkcje **sendmsg()** i **recvmsg()** pozwalają na przekazanie tablicy deskryptorów.

Przykładowy program korzystający z funkcji **sendmsg()** i **recvmsg()** znajduje się w sekcji Przykład: przekazywanie deskryptorów pomiędzy procesami.

Scenariusze dla gniazd: tworzenie aplikacji komunikujących się z klientami IPv4 i IPv6

Sytuacja

Przypuśćmy, że użytkownik jest programistą pracującym w firmie specjalizującej się w aplikacjach gniazd dla serwerów iSeries. Aby uzyskać przewagę nad konkurencją firma zdecydowała się opracować pakiet aplikacji używających rodziny adresów AF_INET6, które będą mogły komunikować się z systemami o adresach IPv4 i IPv6. Zadaniem jest napisanie aplikacji, która będzie przetwarzać żądania zarówno z węzłów IPv4, jak i IPv6. Programista wie, że system OS/400 obsługuje gniazda rodziny adresów AF_INET6 i współdziałanie z gniazdami rodziny adresów AF_INET. Wie także, że funkcję tę realizuje się przy użyciu formatu adresu IPv4 odwzorowanego na adres IPv6. Sekcja Zgodność aplikacji IPv6 z aplikacjami IPv4 zawiera szczegóły dotyczące współdziałania aplikacji IPv6 i IPv4.

Cele scenariusza

Scenariusz ten ma następujące cele:

1. Utworzenie aplikacji serwera, która akceptuje i przetwarza żądania od klientów IPv6 i IPv4.
2. Utworzenie aplikacji klienta, która żąda danych z aplikacji serwera IPv4 lub IPv6.

Wymaganie wstępne

Aby opracować aplikację realizującą te cele, wykonaj następujące czynności:

1. Zainstaluj bibliotekę QSYSINC. Biblioteka ta zawiera pliki nagłówkowe, niezbędne przy kompilowaniu aplikacji używających gniazd.
2. Zainstaluj program licencjonowany C Compiler (5722–CX2).
3. Zainstaluj i skonfiguruj kartę Ethernet 2838. Informacje dotyczące opcji sieci Ethernet zawiera temat Ethernet w Centrum informacyjnym.
4. Skonfiguruj sieć TCP/IP i IPv6.

Szczegóły scenariusza

Poniższa ilustracja przedstawia sieć IPv6, dla której będzie tworzona aplikacja obsługująca żądania od klientów IPv6 i IPv4. Serwer iSeries zawiera program, który będzie nasłuchiwał i przetwarzał żądania od tych klientów.Sieć składa się z dwóch oddzielnych domen, z których w jednej znajdują się wyłącznie klienci IPv4, a w drugiej - wyłącznie klienci IPv6. Nazwą domeny serwera iSeries jest myserver.myco.com. Aplikacja serwera będzie używała rodziny adresów AF_INET6 do obsługi żądań przychodzących, w wywołaniu funkcji **bind()** podając parametr in6addr_any.

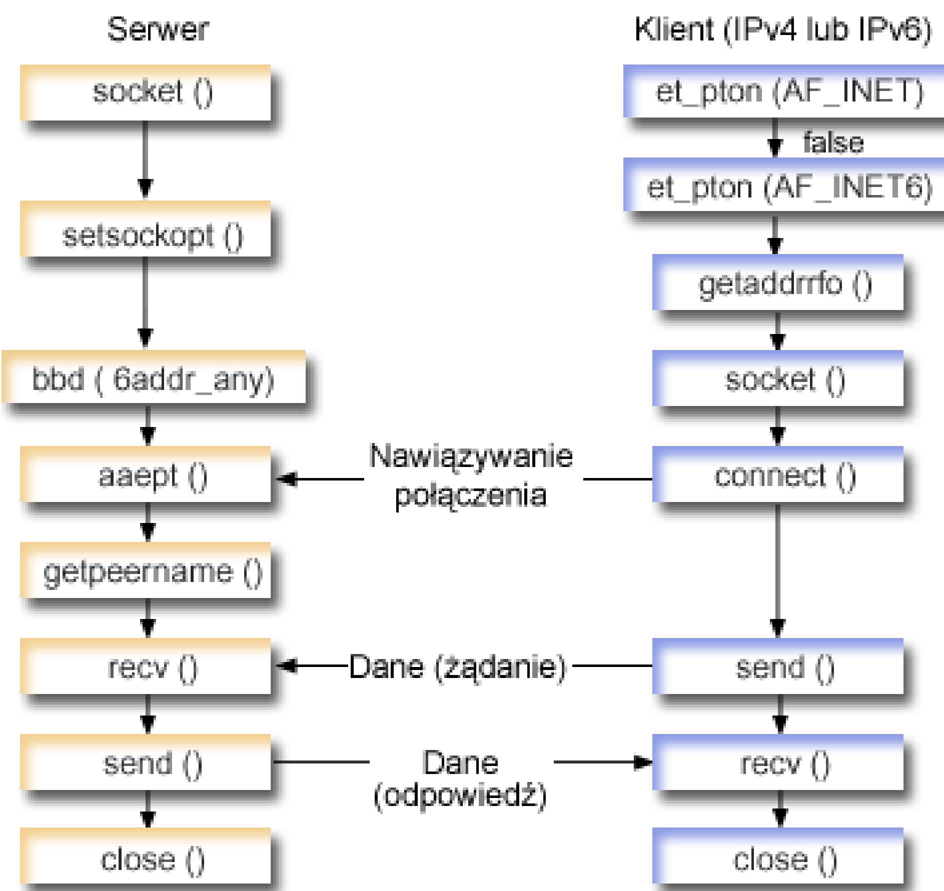
W scenariuszu tym zostały użyte następujące programy przykładowe:

- Przykład: akceptowanie połączeń od klientów IPv6 i IPv4
- Przykład: klient IPv4 lub IPv6

Przykład: akceptowanie połączeń od klientów IPv6 i IPv4

Zadaniem tego programu w układzie serwer/klient jest przyjmowanie żądań z sieci IPv4 (aplikacji gniazd używających rodziny adresów AF_INET) i IPv6 (aplikacji gniazd używających rodziny adresów AF_INET6). Działające obecnie aplikacje gniazd najczęściej używają tylko rodziny adresów AF_INET, dopuszczającej protokoły TCP i UDP; jednak w miarę zwiększania się liczby adresów IPv6 sytuacja ta może ulec zmianie. Program ten może służyć do tworzenia własnych aplikacji, które będą obsługiwać obie rodziny adresów.

Poniższa ilustracja przedstawia sposób działania programu:



Przebieg zdarzeń w gnieździe: aplikacja serwera akceptująca żądania od klientów IPv4 i IPv6

Poniżej opisano poszczególne wywołania funkcji i ich rolę w aplikacji gniazd, która akceptuje żądania od klientów IPv4 i IPv6.

1. Funkcja API **socket()** określa deskryptor gniazda, który tworzy punkt końcowy. Określa również, że dla tego gniazda zostanie użyta rodzina adresów AF_INET6, która obsługuje IPv6 i protokół transportowy TCP (SOCK_STREAM).
2. Funkcja **setsockopt()** umożliwia aplikacji ponowne użycie adresu lokalnego po restarcie serwera, zanim upłynie wymagany czas oczekiwania.
3. Funkcja **bind()** dostarcza unikalnej nazwy gniazda. W tym przykładzie programista ustawia adres na in6addr_any, co (domyślnie) umożliwia nawiązanie połączenia z dowolnego klienta IPv4 lub IPv6, który określi port 3005. (To znaczy, że funkcja bind jest uruchamiana dla obu przestrzeni portów: IPv4 i IPv6).

Uwaga: Jeśli serwer ma obsługiwać wyłącznie klientów IPv6, można użyć opcji gniazda IPV6_ONLY.

4. Funkcja **listen()** umożliwia serwerowi przyjęcie połączenia przychodzącego od klienta. W tym przykładzie programista ustawił wartość kolejki (backlog) na 10. Oznacza to, że system umieści w kolejce pierwsze 10 przychodzących żądań połączenia, a następne odrzuci.
5. Serwer używa funkcji **accept()** do zaakceptowania żądania połączenia przychodzącego. Wywołanie funkcji **accept()** zostanie zablokowane na nieokreślony czas oczekiwania na połączenie przychodzące od klienta IPv4 lub IPv6.
6. Funkcja **getpeername()** zwraca do aplikacji adres klienta. Jeśli jest to klient IPv6, zostanie wyświetlony adres IPv4 odwzorowany na adres IPv6.
7. Funkcja **recv()** odbiera 250 bajtów danych od klienta. W tym przykładzie programista wie, że klient wyśle 250 bajtów danych. Na tej podstawie może użyć opcji gniazda SO_RCVLOWAT i określić, że funkcja **recv()** ma pozostać w uśpieniu do momentu nadejścia całych 250 bajtów danych.
8. Funkcja **send()** odsyła dane do klienta.
9. Funkcja **close()** zamyka wszystkie otwarte deskryptory gniazd.

Przebieg zdarzeń w gnieździe: żądania od klientów IPv4 lub IPv6

Uwaga: Poniższego przykładu klienta można użyć z innymi projektami aplikacji serwera, które akceptują żądania z węzłów IPv4 i IPv6. Można także użyć innych projektów serwera, takich jak opisane w sekcji Przykłady: projekty aplikacji zorientowanych na połączenie.

1. Wywołanie funkcji **inet_pton()** przekształca tekstową postać adresu do binarnej. W tym przykładzie funkcja wywoływana jest dwa razy. Pierwsze wywołanie określa, czy serwer ma poprawny adres AF_INET. Drugie wywołanie funkcji **inet_pton()** określa, czy serwer ma adres z rodziny AF_INET6. Jeśli adres jest liczbowy, funkcja **getaddrinfo()** nie powinna wykonywać rozstrzygania nazwy. W przeciwnym przypadku jest to nazwa hosta, którą należy rozstrzygnąć podczas wywołania funkcji **getaddrinfo()**.
2. Funkcja **getaddrinfo()** pobiera informacje o adresie potrzebne w późniejszych wywołaniach funkcji **socket()** i **connect()**.
3. Funkcja **socket()** zwraca deskryptor gniazda odpowiadający punktowi końcowemu. Instrukcja ta identyfikuje również, na podstawie informacji zwróconych przez funkcję **getaddrinfo()**, rodzinę adresów, typ gniazda i protokół.
4. Funkcja **connect()** nawiązuje połączenie z serwerem niezależnie od tego, czy jest to serwer IPv4 czy IPv6.
5. Funkcja **send()** wysyła do serwera żądanie danych.
6. Funkcja **recv()** odbiera dane z aplikacji serwera.
7. Funkcja **close()** zamyka wszystkie otwarte deskryptory gniazd.

Poniższy kod przykładowy przedstawia aplikację serwera w tym scenariuszu. Aplikacja klienta znajduje się w sekcji Przykład: klient IPv4 lub IPv6. Informacje dotyczące wykorzystania tego kodu zawiera sekcja Informacje dotyczące kodu.

```
/**************************************************************************/
/* Wymagane przez program pliki nagłówkowe.                               */
/**************************************************************************/
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

/**************************************************************************/
/* Stałe używane przez ten program                                        */
/**************************************************************************/
#define SERVER_PORT     3005
#define BUFFER_LENGTH    250
#define FALSE              0

void main()
```

```
{
   /***********************************************************************/
   /* Definicje zmiennych i struktur.                                     */
   /***********************************************************************/
   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];

   /***********************************************************************/
   /* Pętla do/while(FALSE) ułatwia realizację procedur czyszczących w    */
   /* przypadku błędu. Funkcja close() dla poszczególnych deskryptorów    */
   /* gniazd jest uruchamiana jednokrotnie na samym końcu programu.       */
   /***********************************************************************/
   do
   {

      /********************************************************************/
      /* Funkcja socket() zwraca deskryptor gniazda stanowiącego punkt    */
      /* końcowy. Aby przygotować aplikację na odbieranie żądań przycho-  */
      /* dzących w gnieździe, należy pobrać gniazdo dla rodziny AF_INET6. */
      /********************************************************************/
      if ((sd = socket(AF_INET6, SOCK_STREAM, 0)) < 0)
      {
         perror("Niepowodzenie socket()");
         break;
      }

      /********************************************************************/
      /* Funkcja setsockopt() umożliwia ponowne użycie adresu lokalnego   */
      /* przy ponownym uruchomieniu serwera, zanim upłynie wymagany czas  */
      /* oczekiwania.                                                     */
      /********************************************************************/
      if (setsockopt(sd, SOL_SOCKET, SO_REUSEADDR,
                     (char *)&on,sizeof(on)) < 0)
      {
         perror("Niepowodzenie setsockopt(SO_REUSEADDR)");
         break;
      }

      /********************************************************************/
      /* Po utworzeniu deskryptora gniazda funkcja bind() pobiera         */
      /* unikalną nazwę gniazda. W tym przykładzie użytkownik ustawia     */
      /* adres na in6addr_any, który (domyślnie) umożliwia nawiązywanie   */
      /* połączeń z dowolnego klienta IPv4 lub IPv6, który określi port   */
      /* 3005. Oznacza to, że funkcja bind jest uruchamiana dla stosów    */
      /* IPv4 i IPv6. Zachowanie to można w razie potrzeby zmodyfikować   */
      /* za pomocą opcji gniazda IPV6_V6ONLY na poziomie IPPROTO_IPV6.    */
      /********************************************************************/
      memset(&serveraddr, 0, sizeof(serveraddr));
      serveraddr.sin6_family = AF_INET6;
      serveraddr.sin6_port   = htons(SERVER_PORT);
      /********************************************************************/
      /* Uwaga: Aplikacje używają struktury in6addr_any w podobny sposób, */
      /* jak makra INADDR_ANY w protokole IPv4. Stała symboliczna         */
      /* IN6ADDR_ANY_INIT również istnieje, ale może zostać użyta tylko   */
      /* do zainicjowania struktury in6_addr podczas deklarowania (nie    */
      /* podczas przypisywania).                                          */
      /********************************************************************/
      serveraddr.sin6_addr   = in6addr_any;
      /********************************************************************/
      /* Uwaga: pozostałe pola w strukturze sockaddr_in6 nie są obecnie   */
      /* obsługiwane i aby zapewnić zgodność z nowszymi wersjami, należy  */
      /* je ustawić na 0.                                                 */
      /********************************************************************/
```

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

/********************************************************************/
/* Funkcja listen() umożliwia serwerowi przyjęcie połączeń          */
/* przychodzących od klienta. W tym przykładzie kolejka (backlog)   */
/* ma wartość 10. Oznacza to, że system umieści w kolejce pierwsze  */
/* 10 przychodzących żądań połączenia, a następne będzie je         */
/* odrzucał.                                                        */
/********************************************************************/
if (listen(sd, 10) < 0)
{
   perror("Niepowodzenie listen()");
   break;
}

printf("Gotowy do obsługi klienta (connect()).\n");

/********************************************************************/
/* Serwer używa funkcji accept() do zaakceptowania połączenia       */
/* przychodzącego. Wywołanie funkcji accept() zostanie zablokowane  */
/* na nieokreślony czas oczekiwania na połączenie przychodzące od   */
/* klienta IPv4 lub IPv6                                            */
/********************************************************************/
if ((sdconn = accept(sd, NULL, NULL)) < 0)
{
   perror("Niepowodzenie accept()");
   break;
}
else
{
   /*****************************************************************/
   /* Wyświetla adres klienta. Jeśli jest to klient IPv6, zostanie  */
   /* wyświetlony adres IPv4 odwzorowany na adres IPv6.             */
   /*                                                               */
   /*****************************************************************/
   getpeername(sdconn, (struct sockaddr *)&clientaddr, &addrlen);
   if(inet_ntop(AF_INET6, &clientaddr.sin6_addr, str, sizeof(str))) {
      printf("Adresem klienta jest %s\n", str);
      printf("Portem klienta jest %d\n", ntohs(clientaddr.sin6_port));
   }
}

/********************************************************************/
/* W tym przykładzie wiadomo, że klient wyśle 250 bajtów danych.    */
/* Dzięki temu można użyć opcji gniazda SO_RCVLOWAT i określić, że  */
/* funkcja recv() ma nie wychodzić z uśpienia, dopóki nie zostanie  */
/* odebrane wszystkie 250 bajtów danych.                            */
/********************************************************************/
if (setsockopt(sdconn, SOL_SOCKET, SO_RCVLOWAT,
               (char *)&rcdsize,sizeof(rcdsize)) < 0)
{
   perror("Niepowodzenie setsockopt(SO_RCVLOWAT)");
   break;
}

/********************************************************************/
/* Odebranie 250 bajtów od klienta                                  */
/********************************************************************/
rc = recv(sdconn, buffer, sizeof(buffer), 0);
if (rc < 0)
```

```
      {
         perror("Niepowodzenie recv()");
         break;
      }

      printf("Otrzymano dane, bajtów: %d\n", rc);
      if (rc == 0 ||
          rc < sizeof(buffer))
      {
         printf("Klient zamknął połączenie przed wysłaniem\n");
         printf("wszystkich danych\n");
         break;
      }

      /********************************************************************/
      /* Odesłanie danych do klienta                */
      /********************************************************************/
      rc = send(sdconn, buffer, sizeof(buffer), 0);
      if (rc < 0)
      {
         perror("Niepowodzenie send()");
         break;
      }

      /********************************************************************/
      /* Zakończenie programu                                             */
      /********************************************************************/

   } while (FALSE);

   /***********************************************************************/
   /* Zamknięcie wszystkich otwartych deskryptorów gniazd                 */
   /***********************************************************************/
   if (sd != -1)
      close(sd);
   if (sdconn != -1)
      close(sdconn);
}
```

Przykład: klient IPv4 lub IPv6

Tego przykładowego programu klienta można użyć razem z aplikacją serwera, która akceptuje żądania od klientów IPv4 i IPv6.

```
/**************************************************************************/
/* Oto klient IPv4 lub IPv6.                                              */
/**************************************************************************/

/**************************************************************************/
/* Wymagane przez program pliki nagłówkowe.                               */
/**************************************************************************/
#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>

/**************************************************************************/
/* Stałe używane przez ten program                                        */
/**************************************************************************/
#define BUFFER_LENGTH    250
#define FALSE              0
#define SERVER_NAME     "NazwaSerweraHosta"

/* Przekaż 1 parametr, który albo jest     */
```

```
/* adresem, albo nazwą hosta serwera lub   */
/* ustaw nazwę serwera zmienną #define     */
/* SERVER_NAME.                            */
void main(int argc, char *argv[])
{
   /*********************************************************************/
   /* Definicje zmiennych i struktur.                                   */
   /*********************************************************************/
   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;

   /*********************************************************************/
   /* Pętla do/while(FALSE) ułatwia realizację procedur czyszczących w  */
   /* przypadku błędu. Funkcja close() dla deskryptora gniazda jest     */
   /* wykonywana tylko raz na końcu programu i opróżnia listę adresów.  */
 /*********************************************************************/
   do
   {
      /******************************************************************/
      /* Jeśli został przekazany argument, należy go użyć jako nazwy    */
      /* serwera, w przeciwnym razie należy użyć zmiennej określonej    */
      /* w makrze #define znajdującym się na początku programu.         */
      /******************************************************************/
      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;
      /******************************************************************/
      /* Sprawdza, czy został dostarczony adres serwera za pomocą funkcji */
      /* inet_pton(), która przekształca tekstową postać adresu do      */
      /* binarnej. Jeśli adres jest liczbowy, funkcja getaddrinfo() nie  */
      /* powinna wykonywać rozstrzygania nazwy.                          */
      /******************************************************************/
      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) /* poprawny adres tekstowy IPv6? */
         {

            hints.ai_family = AF_INET6;
            hints.ai_flags |= AI_NUMERICHOST;
         }
      }
      /******************************************************************/
      /* Pobranie informacji o serwerze za pomocą funkcji getaddrinfo().  */
      /******************************************************************/
      rc = getaddrinfo(server, servport, &hints, &res);
      if (rc != 0)
      {
         printf("Hosta nie znaleziono --> %s\n", gai_strerror(rc));
         if (rc == EAI_SYSTEM)
            perror("Niepowodzenie getaddrinfo()");
```

```
   break;
}

/*****************************************************************/
/* Funkcja socket() zwraca deskryptor gniazda stanowiącego punkt    */
/* końcowy. Instrukcja ta identyfikuje również rodzinę adresów,    */
/* typ gniazda i protokół na podstawie informacji zwróconych przez  */
/* funkcję getaddrinfo().                                           */
/*****************************************************************/
sd = socket(res->ai_family, res->ai_socktype, res->ai_protocol);
if (sd < 0)
{
   perror("Niepowodzenie socket()");
   break;
}
/*****************************************************************/
/* Aby nawiązać połączenie z serwerem, zostanie użyta funkcja       */
/* connect().                                                       */
/*****************************************************************/
rc = connect(sd, res->ai_addr, res->ai_addrlen);
if (rc < 0)
{
   /**************************************************************/
   /* Uwaga: res jest dowiązaną listą adresów znalezionych dla     */
   /* serwera. Jeśli funkcja connect() nie powiedzie się przy      */
   /* pierwszym, mogą zostać wypróbowane kolejne adresy na liście.  */
   /**************************************************************/
   perror("Niepowodzenie connect()");
   break;
}

/*****************************************************************/
/* Wysłanie 250 bajtów znaków 'a' do serwera                        */
/*****************************************************************/
memset(buffer, 'a', sizeof(buffer));
rc = send(sd, buffer, sizeof(buffer), 0);
if (rc < 0)
{
   perror("Niepowodzenie send()");
   break;
}

/*****************************************************************/
/* W tym przykładzie wiadomo, że serwer odpowie wysłaniem tych      */
/* samych 250 bajtów, które wysłaliśmy. Ponieważ wiadomo, że        */
/* zostanie odesłanych 250 bajtów, można użyć opcji gniazda         */
/* SO_RCVLOWAT, uruchomić pojedynczą funkcję recv() i pobrać        */
/* wszystkie dane.                                                  */
/*                                                                  */
/* Użycie opcji SO_RCVLOWAT zostało już pokazane w przykładzie      */
/* serwera, dlatego tutaj użyjemy innej metody. Ponieważ te 250     */
/* bajtów danych może być przysłanych w oddzielnych pakietach,      */
/* będziemy wielokrotnie uruchamiali funkcję recv(), dopóki         */
/* nie odbierzemy wszystkich 250 bajtów.                            */
/*****************************************************************/
while (bytesReceived < BUFFER_LENGTH)
{
   rc = recv(sd, & buffer[bytesReceived],
             BUFFER_LENGTH - bytesReceived, 0);
   if (rc < 0)
   {
      perror("Niepowodzenie recv()");
      break;
   }
   else if (rc == 0)
   {
      printf("Serwer zamknął połączenie\n");
```

```
            break;
         }

         /**************************************************************/
         /* Zwiększenie liczby otrzymanych dotychczas bajtów           */
         /**************************************************************/
         bytesReceived += rc;
      }

   } while (FALSE);

   /***********************************************************************/
   /* Zamknięcie wszystkich otwartych deskryptorów gniazd                 */
   /***********************************************************************/
   if (sd != -1)
      close(sd);
   /***********************************************************************/
   /* Zwolnienie wszystkich rezultatów działania funkcji getaddrinfo      */
   /***********************************************************************/
   if (res != NULL)
      freeaddrinfo(res);
}
```

Zalecenia dotyczące projektowania aplikacji używających gniazd

Aby projektować aplikacje używające gniazd, należy określić wymagania funkcjonalne, cele i potrzeby tej aplikacji. Należy również rozważyć wymagania dotyczące wydajności oraz wpływ zasobów systemowych na aplikację. Poniższa lista zaleceń może pomóc rozwiązać te kwestie i wskazać efektywniejsze metody wykorzystania gniazd i projektowania aplikacji używających gniazd:

Tabela 19. Projektowanie aplikacji używających gniazd

Zalecenie	Uzasadnienie	Najlepsze zastosowanie
Użyj asynchronicznych operacji we/wy	Zamiast konwencjonalnego modelu opartego na funkcji select() preferowane jest użycie asynchronicznych operacji we/wy w modelu serwera z obsługą wątków. Więcej informacji o korzyściach płynących z zastosowania asynchronicznych operacji we/wy zawiera sekcja Asynchroniczne operacje we/wy. Przykład programu używającego asynchronicznych operacji we/wy znajduje się w sekcji Przykład: korzystanie z asynchronicznych operacji we/wy.	Aplikacje serwera używające gniazd, przeznaczone do współbieżnej obsługi wielu klientów.
Używając asynchronicznych operacji we/wy, dostosuj liczbę wątków w procesie do liczby optymalnej dla liczby klientów, którzy mają być przetwarzani.	W przypadku zbyt małej liczby wątków niektórzy klienci mogą przekroczyć czas oczekiwania na obsługę. W przypadku zbyt dużej liczby wątków niektóre zasoby systemowe nie będą wykorzystywane efektywnie. **Uwaga:** Lepiej, gdy wątków jest za dużo niż za mało.	Aplikacje używające gniazd, korzystające z asynchronicznych operacji we/wy.
Projektuj aplikacje używające gniazd tak, aby uniknąć używania postflag dla wszystkich operacji rozpoczynających asynchroniczne operacje we/wy.	Pozwala to uniknąć dodatkowego obciążenia związanego z przejściem do portu zakończenia, jeśli operacja została wykonana synchronicznie.	Aplikacje używające gniazd, korzystające z asynchronicznych operacji we/wy.

Tabela 19. Projektowanie aplikacji używających gniazd (kontynuacja)

Korzystaj z funkcji **send()** i **recv()** zamiast **read()** i **write()**.	Funkcje **send()** i **recv()** mają nieco poprawioną wydajność i możliwości obsługi serwisowej w stosunku do funkcji **read()** i **write()**.	Dowolny program używający gniazd, który rozróżnia używanie deskryptorów gniazd od deskryptorów plików.
Używaj dla gniazda opcji odbierania SO_RCVLOWAT, aby uniknąć zapętlenia podczas operacji odbierania, dopóki nie zostaną przysłane wszystkie dane.	Dzięki temu aplikacja może oczekiwać na minimalną ilość danych do odebrania przez gniazdo przed spełnieniem warunku dla zablokowanej operacji odbioru.	Dowolna aplikacja używająca gniazd, która odbiera dane.
Używaj opcji MSG_WAITALL, aby uniknąć zapętlenia podczas operacji odbierania, dopóki nie zostaną przysłane wszystkie dane.	Dzięki temu aplikacja może oczekiwać na całą zawartość buforu operacji odbierania, zanim zostanie spełniony warunek zablokowanej operacji odbioru.	Dowolna aplikacja używająca gniazd, która odbiera dane i wie z góry, ile danych ma oczekiwać.
Używaj funkcji **sendmsg()** i **recvmsg()** zamiast funkcji **givedescriptor()** i **takedescriptor()**.	Zalety tych funkcji opisano w sekcji Przekazywanie deskryptorów pomiędzy procesami – funkcje sendmsg() i recvmsg(). W sekcji Przykład: przekazywanie deskryptorów pomiędzy procesami znajduje się przykład programu używającego funkcji **sendmsg()** i **recvmsg()**.	Dowolna aplikacja używająca gniazd, która przekazuje deskryptory gniazd lub plików pomiędzy procesami.
Używając funkcji **select()**, próbuj unikać dużej liczby deskryptorów w zestawie operacji odczytu, zapisu lub w zestawie wyjątków. **Uwaga:** W przypadku dużej liczby deskryptorów używanych z przetwarzaniem funkcji **select()**, patrz zalecenie dotyczące używania asynchronicznych operacji we/wy.	W przypadku dużej liczby deskryptorów dla zestawu operacji odczytu, zapisu lub w zestawie wyjątków, przy każdym wywołaniu funkcji **select()** wykonywanych jest wiele powtarzających się czynności. Po wykonaniu funkcji **select()** należy wykonać rzeczywistą funkcję gniazda, na przykład funkcję odczytu, zapisu lub akceptacji. Funkcje API dla asynchronicznych operacji we/wy łączą powiadomienie o zdarzeniu z rzeczywistą operacją we/wy.	Aplikacje o dużej (> 50) liczbie aktywnych deskryptorów dla funkcji **select()**.
Zapisz zestawy operacji odczytu zapisu lub zestawy wyjątków przed wywołaniem funkcji **select()**, aby uniknąć odbudowywania zestawów podczas każdego wywołania funkcji **select()**.	Ta czynność eliminuje dodatkowe czynności związane z przebudowywaniem zestawów operacji odczytu, zapisu i zestawów wyjątków za każdym razem, gdy planowane jest wywołanie funkcji **select()**. Sekcja Przykład: nieblokujące operacje we/wy i funkcja select() zawiera przykład programu wykorzystującego funkcję **select()**.	Dowolna aplikacja używająca gniazd, w której wykorzystuje się funkcję **select()** z dużą liczbą deskryptorów aktywnych dla przetwarzania odczytu, zapisu lub wyjątków.
Nie używaj funkcji **select()** jako zegara. Zamiast niej używaj funkcji **sleep()**. **Uwaga:** Jeśli granulacja funkcji **sleep()** nie jest odpowiednia, może być konieczne użycie funkcji **select()** jako zegara. W takim przypadku ustaw maksymalną liczbę deskryptorów na 0, a zestawy operacji odczytu, zapisu i zestawy wyjątków na NULL.	Zapewni to lepsze odpowiedzi zegara i zmniejszy obciążenie systemu.	Dowolna aplikacja używająca gniazd, w której funkcja **select()** jest stosowana wyłącznie jako zegar.

Tabela 19. Projektowanie aplikacji używających gniazd (kontynuacja)

Jeśli aplikacja używająca gniazd ma zwiększoną maksymalną liczbę deskryptorów gniazd i plików dozwolonych dla pojedynczego procesu za pomocą funkcji DosSetRelMaxFH() oraz jeśli w tej samej aplikacji używana jest funkcja **select()**, zwróć uwagę na wpływ, jaki ta nowa wartość maksymalna ma na wielkość zestawów operacji odczytu, zapisu i zestawów wyjątków używanych w przetwarzaniu funkcji **select()**.	Przydzielenie deskryptora spoza zakresu zestawów operacji odczytu, zapisu i zestawów wyjątków, określonego przez wartość FD_SETSIZE, może spowodować nadpisanie i uszkodzenie pamięci masowej. Wielkość zestawów powinna być przynajmniej na tyle duża, aby obsłużyć maksymalną liczbę deskryptorów ustawionych dla procesu i maksymalną liczbę deskryptorów określoną dla funkcji **select()**.	Dowolna aplikacja lub proces, gdzie używa się funkcji DosSetRelMaxFH() i **select()**.
Ustaw wszystkie deskryptory gniazd w zestawach operacji odczytu lub zapisu jako nieblokujące. Kiedy deskryptor staje się dostępny do odczytu lub zapisu, zapętla się i przyjmuje lub wysyła wszystkie dane dopóki nie zostanie zwrócona wartość EWOULDBLOCK. W sekcji Przykład: nieblokujące operacje we/wy i funkcja select() znajduje się przykład programu używającego funkcji **select()**.	Pozwoli to zminimalizować liczbę wywołań funkcji **select()**, kiedy dane są wciąż dostępne do przetwarzania lub odczytu dla danego deskryptora.	Dowolna aplikacja używająca gniazd, w której stosowana jest funkcja **select()**.
Wykorzystując przetwarzanie funkcji **select()**, określ tylko niezbędne zestawy.	Większość aplikacji nie potrzebuje określania zestawu wyjątków ani zestawu operacji zapisu.	Dowolna aplikacja używająca gniazd, w której stosowana jest funkcja **select()**.
Używaj funkcji API GSKit zamiast funkcji API SSL.	Funkcje API Global Secure Toolkit (GSKit), jak też OS/400 SSL_, umożliwiają tworzenie chronionych aplikacji dla rodziny adresów AF_INET lub AF_INET6, używających gniazd SOCK_STREAM. Ponieważ Funkcje API GSKit są obsługiwane przez wszystkie platformy IBM @server, do tworzenia chronionych aplikacji zaleca się używanie tego zestawu. Funkcje API SSL_ są rodzime tylko w systemie OS/400.	Każda aplikacja używająca gniazd, która wymaga obsługi przetwarzania SSL/TLS.
Nie używaj sygnałów.	Obciążenie związane z sygnałami jest kosztowne (na wszystkich platformach, nie tylko dla iSeries). Lepiej zaprojektować aplikację, która będzie używać asynchronicznych operacji we/wy lub funkcji **select()**.	Każdy programista, który zastanawia się nad wykorzystaniem sygnałów w swojej aplikacji używającej gniazd.
W miarę możliwości używaj procedur niezależnych od protokołu, takich jak **inet_ntop()**, **inet_pton()**, **getaddrinfo()** i **getnameinfo()**.	Warto używać tych funkcji API (zamiast **inet_ntoa()**, **inet_addr()**, **gethostbyname()** i **gethostbyaddr()**) nawet wtedy, gdy aplikacja nie ma obsługiwać IPv6, aby ułatwić późniejszą migrację.	Dowolna aplikacja AF_INET lub AF_INET6 korzystająca z procedur sieciowych.
Użyj sockaddr_storage do zadeklarowania obszaru pamięci dla dowolnego adresu z tej rodziny adresów.	Upraszcza pisanie kodu przenośnego pomiędzy wieloma rodzinami adresów i platformami. Deklaruje wystarczającą ilość pamięci, aby pomieściła największą rodzinę adresów i zapewnia poprawne wyrównanie adresu.	Dowolna aplikacja używająca gniazd, która przechowuje adresy.

Przykłady: projekty aplikacji używających gniazd

Poniższe przykłady zawierają wiele programów ilustrujących zaawansowane koncepcje dotyczące gniazd. Można ich użyć do tworzenia własnych aplikacji, które będą realizowały podobne zadania. Przykładom towarzyszą ilustracje i lista wywołań, które ilustrują przebieg zdarzeń w poszczególnych aplikacjach. Można użyć narzędzia Xsocket do interaktywnego wypróbowania niektórych funkcji API w tych programach lub dokonać odpowiednich zmian, aby przystosować je do środowiska, w którym będą pracować.

- Przykłady: projekty aplikacji zorientowanych na połączenie
- Przykład: nawiązywanie chronionych połączeń
- Przykład: procedury sieciowe obsługujące ochronę wątków i używające funkcji gethostbyaddr_r()
- Przykład: nieblokujące operacje we/wy i funkcja select()
- Przykład: używanie sygnałów z blokującymi funkcjami API gniazd
- Przykład: użycie rozsyłania grupowego dla rodziny adresów AF_INET
- Przykład: odpytywanie i aktualizacja serwera DNS
- Przykład: przesyłanie danych pliku za pomocą funkcji send_file() i accept_and_recv()

Przykłady: projekty aplikacji zorientowanych na połączenie

W systemie iSeries jest wiele możliwości realizacji zorientowanego na połączenie serwera gniazd. Do utworzenia własnych programów zorientowanych na połączenie można użyć poniższych programów przykładowych. Mimo iż możliwe są inne projekty serwerów używających gniazd, poniższe przykłady są używane najczęściej:

Serwer iteracyjny

W przykładzie serwera iteracyjnego pojedyncze zadanie serwera obsługuje wszystkie połączenia przychodzące, a wszystkie dane przepływają do zadań klienta. Po zakończeniu działania funkcji API **accept()** serwer obsługuje całą transakcję. Jest to najprostszy serwer, ale wiążą się z nim pewne problemy. Podczas gdy serwer obsługuje żądanie od określonego klienta, inne mogą próbować się z nim połączyć. Żądania te zapełniają kolejkę (backlog) **listen()** i niektóre zostaną w pewnym momencie odrzucone.

Wszystkie pozostałe przykłady to projekty serwerów współbieżnych. W tych projektach system używa wielu zadań i wątków do obsługi przychodzących żądań połączenia. Serwerów współbieżnych używa się, gdy wielu klientów może łączyć się z nimi jednocześnie.

W przypadku wielu współbieżnych klientów w sieci zaleca się użycie funkcji API gniazd obsługujących asynchroniczne operacje we/wy. Funkcje te zapewniają największą wydajność w sieciach z wieloma klientami działającymi współbieżnie. Sekcja Asynchroniczne operacje we/wy opisuje sposób działania tych funkcji API. Przykładowy program korzystający z tych funkcji API znajduje się w sekcji Przykład: używanie asynchronicznych operacji we/wy.

Serwer spawn() i proces roboczy spawn()

Przykład serwera spawn() i procesu roboczego spawn() używa funkcji API spawn() do utworzenia nowego zadania, które obsłuży każde żądanie przychodzące. Po zakończeniu działania funkcji spawn() serwer może czekać przy funkcji API **accept()** na odebranie następnego połączenia przychodzącego.

Jedynym problemem związanym z tym projektem serwera jest nakład pracy związany z tworzeniem nowego zadania przy każdym połączeniu. Można tego uniknąć, używając zadań prestartu. Zamiast tworzyć nowe zadanie dla każdego połączenia, połączenie przychodzące jest przekazywane do zadania, które jest już aktywne. Wszystkie pozostałe przykłady w tym temacie używają zadań prestartu.

Serwer sendmsg() i proces roboczy recvmsg()

Serwer sendmsg() i proces roboczy recvmsg() przekazują połączenie przychodzące do zadania procesu roboczego (klienta). Przy pierwszym uruchomieniu zadania serwera prestartuje ono wszystkie zadania procesu roboczego.

Wiele serwerów accept() i wiele procesów roboczych accept()

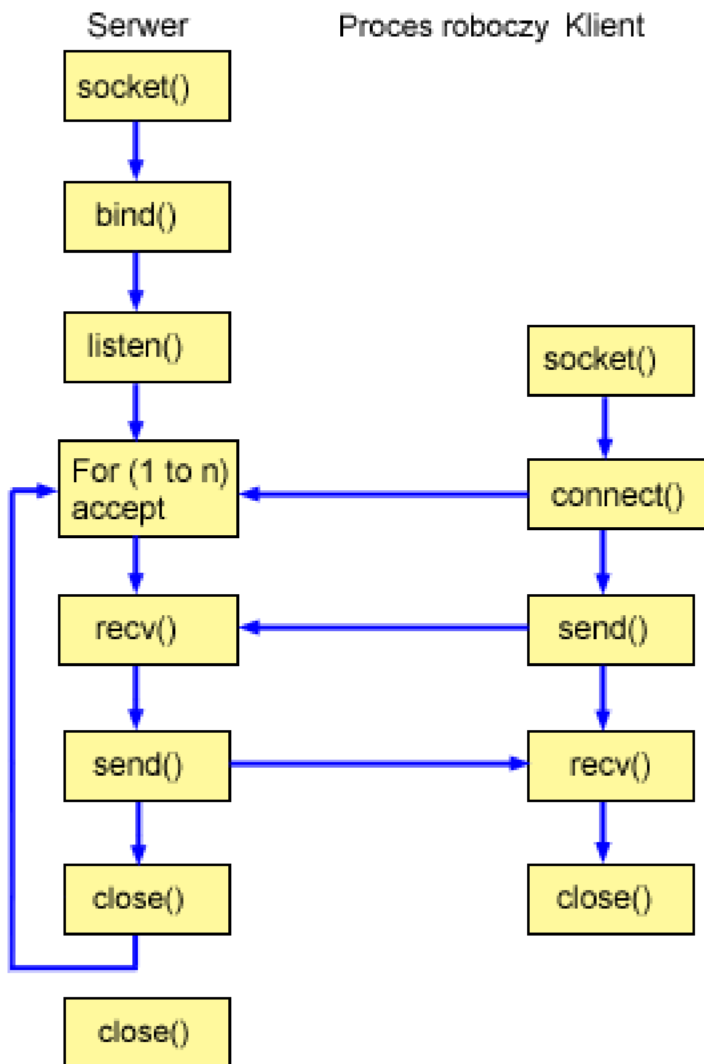
W poprzednich przykładach proces roboczy nie wykonywał żadnych czynności, dopóki serwer nie odebrał żądania połączenia przychodzącego. Przykład wielu serwerów accept() i wielu procesów roboczych accept()

czyni z każdego zadania procesu roboczego serwer iteracyjny. Zadanie serwera nadal wywołuje funkcje API **socket()**, **bind()** i **listen()**. Po zakończeniu działania funkcji **listen()** serwer tworzy poszczególne zadania procesów roboczych i daje każdemu z nich gniazdo do nasłuchu. Następnie wszystkie zadania procesów roboczych wywołują funkcję API **accept()**. Gdy klient próbuje połączyć się z serwerem, jedno wywołanie funkcji **accept()** kończy działanie i ten proces roboczy obsługuje połączenie.

Wszystkie poniższe przykłady zostały opracowane dla podstawowego modelu klienta. Szczegóły zawiera sekcja Przykład: ogólny program klienta.

Przykład: pisanie programu serwera iteracyjnego

Poniżej znajduje się przykład pojedynczego zadania serwera, które obsługuje wszystkie połączenia przychodzące. Po zakończeniu działania funkcji API **accept()** serwer obsługuje całą transakcję. Rysunek przedstawia, jak zadania serwera i klienta współdziałają ze sobą, gdy w systemie używany jest program serwera iteracyjnego. W sekcji Przykład: ogólny program klienta znajduje się przykład kodu dla typowego zadania klienta, którego można użyć z niniejszym przykładem.



Przebieg zdarzeń w gnieździe: serwer iteracyjny

Poniżej wyjaśniono sekwencję wywołań funkcji gniazd, przedstawionych na powyższej ilustracji. Opisano także relacje pomiędzy aplikacją serwera a aplikacją procesu roboczego. Każdy zbiór wywołań zawiera odsyłacze do uwag dotyczących użycia poszczególnych funkcji API. Aby uzyskać szczegółowe informacje dotyczące użycia tych funkcji

API, należy użyć poniższych odsyłaczy. Opis funkcji wykonywanych kolejno przez klienta zawiera sekcja Przykład: ogólny program klienta. Poniżej opisano kolejność wywołań funkcji w przykładowej aplikacji serwera iteracyjnego:

1. Funkcja **socket()** zwraca deskryptor gniazda odpowiadający punktowi końcowemu. Instrukcja ta informuje również, że dla tego gniazda zostanie użyta rodzina adresów INET (Internet Protocol) z transportem TCP transport (SOCK_STREAM).
2. Po utworzeniu deskryptora gniazda funkcja **bind()** pobiera unikalną nazwę gniazda.
3. Funkcja **listen()** umożliwia serwerowi przyjęcie połączenia przychodzącego od klienta.
4. Serwer używa funkcji **accept()** do zaakceptowania żądania połączenia przychodzącego. Wywołanie funkcji **accept()** zostanie zablokowane na nieokreślony czas oczekiwania na połączenie przychodzące.
5. Funkcja **recv()** odbiera dane z aplikacji klienta.
6. Funkcja **send()** odsyła dane do klienta.
7. Funkcja **close()** zamyka wszystkie otwarte deskryptory gniazd.

```
/**************************************************************************/
/* Aplikacja serwera iteracyjnego                                         */
/**************************************************************************/
#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;

   /*************************************************/
   /* Jeśli podano argument, użyj go do sterowania  */
   /* liczbą połączeń przychodzących                */
   /*************************************************/
   if (argc >= 2)
      num = atoi(argv[1]);
   else
      num = 1;

   /*************************************************/
   /* Utwórz gniazdo strumienia AF_INET do          */
   /* odbierania połączeń przychodzących            */
   /*************************************************/
   listen_sd = socket(AF_INET, SOCK_STREAM, 0);
   if (listen_sd < 0)
   {
      perror("Niepowodzenie socket()");
      exit(-1);
   }

   /*************************************************/
   /* Umożliwia ponowne użycie deskryptora gniazda  */
   /*************************************************/
   rc = setsockopt(listen_sd,
                   SOL_SOCKET,  SO_REUSEADDR,
                   (char *)&on, sizeof(on));
   if (rc < 0)
   {
      perror("Niepowodzenie setsockopt()");
      close(listen_sd);
      exit(-1);
   }
```

```
/*************************************************/
/* Powiąż gniazdo                                */
/*************************************************/
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("Niepowodzenie bind()");
   close(listen_sd);
   exit(-1);
}

/*************************************************/
/* Ustawia nasłuchiwanie parametru backlog.      */
/*************************************************/
rc = listen(listen_sd, 5);
if (rc < 0)
{
   perror("Niepowodzenie listen()");
   close(listen_sd);
   exit(-1);
}

/*************************************************/
/* Poinformuj użytkownika o tym, że              */
/* serwer jest gotowy                            */
/*************************************************/
printf("Serwer jest gotowy\n");

/*************************************************/
/* Przejdź przez pętlę raz dla każdego połączenia*/
/*************************************************/
for (i=0; i < num; i++)
{
   /**********************************************/
   /* Oczekiwanie na połączenie przychodzące     */
   /**********************************************/
   printf("Iteracja: %d\n", i+1);
   printf("  oczekuje na accept()\n");
   accept_sd = accept(listen_sd, NULL, NULL);
   if (accept_sd < 0)
   {
      perror("Niepowodzenie accept()");
      close(listen_sd);
      exit(-1);
   }
   printf(" accept zakończone powodzeniem\n");

   /**********************************************/
   /* Odebranie komunikatu od klienta            */
   /**********************************************/
   printf("Oczekiwanie na przysłanie komunikatu od klienta\n");
   rc = recv(accept_sd, buffer, sizeof(buffer), 0);
   if (rc <= 0)
   {
      perror("Niepowodzenie recv()");
      close(listen_sd);
      close(accept_sd);
      exit(-1);
   }
   printf("  <%s>\n", buffer);

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

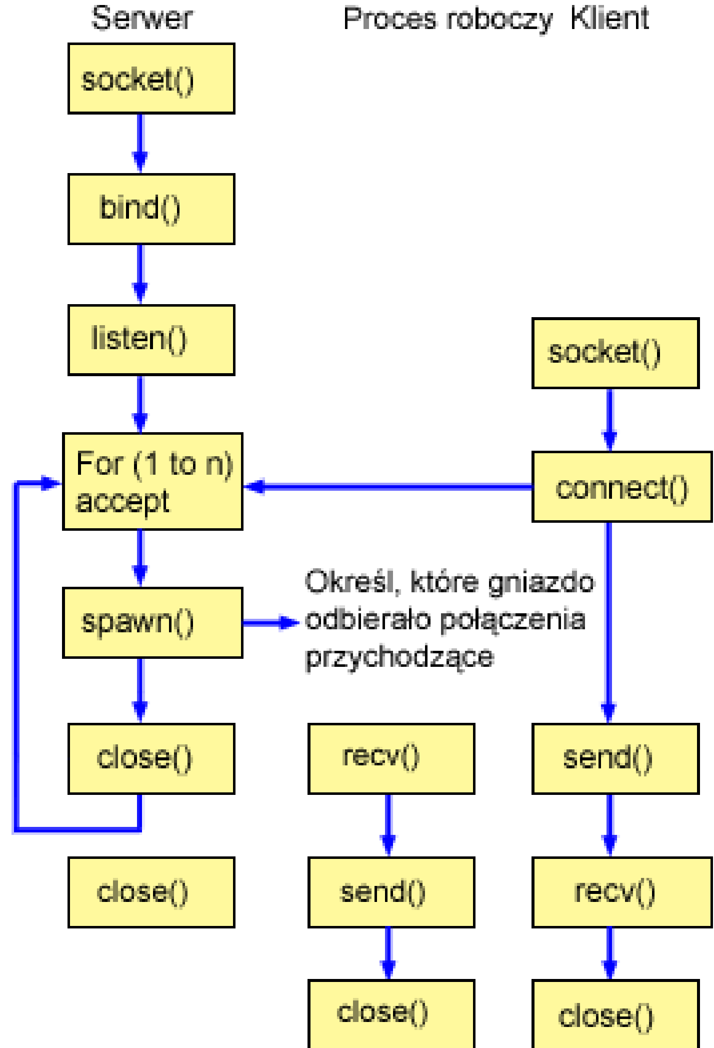
```
      /* Odesłanie danych do klienta                */
      /**********************************************/
      printf("odesłanie\n");
      len = rc;
      rc = send(accept_sd, buffer, len, 0);
      if (rc <= 0)
      {
         perror("Niepowodzenie send()");
         close(listen_sd);
         close(accept_sd);
         exit(-1);
      }

      /**********************************************/
      /* Zamknięcie połączenia przychodzącego       */
      /**********************************************/
      close(accept_sd);
   }

   /*************************************************/
   /* Zamyka gniazdo nasłuchujące.                  */
   /*************************************************/
   close(listen_sd);
}
```

Przykład: używanie funkcji API spawn() do tworzenia procesów potomnych

Niniejszy przykład pokazuje, jak program serwera może użyć funkcji API **spawn()** do utworzenia procesu potomnego, który dziedziczy deskryptor gniazda po procesie macierzystym. Zadanie serwera czeka na połączenie przychodzące, a następnie wywołuje funkcję **spawn()** w celu utworzenia procesów potomnych do obsługi połączenia. W ramach wywołania funkcji **spawn()** proces potomny dziedziczy następujące atrybuty:

- deskryptory gniazda i pliku,
- maskę sygnału,
- wektor działania wywołanego sygnałem,
- zmienne środowiskowe.

Poniższy rysunek przedstawia, jak zadania serwera, procesu roboczego i klienta współdziałają ze sobą, gdy w systemie używany jest program serwera **spawn()**.



Przebieg zdarzeń w gnieździe: serwer używający funkcji spawn() do przyjmowania i przetwarzania żądań
Poniżej wyjaśniono sekwencję wywołań funkcji gniazd, przedstawionych na powyższej ilustracji. Opisano także relacje pomiędzy aplikacją serwera a przykładowym procesem roboczym. Każdy zbiór wywołań zawiera odsyłacze do uwag dotyczących użycia poszczególnych funkcji API. Aby uzyskać szczegółowe informacje dotyczące użycia tych funkcji API, należy użyć poniższych odsyłaczy. W sekcji Przykład: tworzenie serwera używającego funkcji spawn() do utworzenia procesu potomnego za pomocą wywołania funkcji **spawn()** wykorzystano następujące wywołania funkcji gniazd:

1. Funkcja **socket()** zwraca deskryptor gniazda odpowiadający punktowi końcowemu. Instrukcja ta informuje również, że dla tego gniazda zostanie użyta rodzina adresów INET (Internet Protocol) z transportem TCP transport (SOCK_STREAM).
2. Po utworzeniu deskryptora gniazda funkcja **bind()** pobiera unikalną nazwę gniazda.
3. Funkcja **listen()** umożliwia serwerowi przyjęcie połączenia przychodzącego od klienta.
4. Serwer używa funkcji **accept()** do zaakceptowania żądania połączenia przychodzącego. Wywołanie funkcji **accept()** zostanie zablokowane na nieokreślony czas oczekiwania na połączenie przychodzące.
5. Funkcja **spawn()** inicjuje parametry zadania roboczego do obsługi żądań przychodzących. W tym przykładzie deskryptor gniazda nowego połączenia jest p programie potomnym odwzorowywany na deskryptor 0.
6. W tym przykładzie pierwsze wywołanie funkcji **close()** zamyka deskryptor gniazda nasłuchującego. Drugie wywołanie funkcji **close()** kończy zaakceptowane gniazdo.

Przebieg zdarzeń w gnieździe: zadanie procesu roboczego utworzone przy użyciu funkcji spawn()
W sekcji Przykład: umożliwianie przekazania bufora danych do zadania procesu roboczego użyto następujących wywołań funkcji:

1. Po wywołaniu w serwerze funkcji **spawn()** funkcja **recv()** odbiera dane nadchodzące z połączenia przychodzącego.
2. Funkcja **send()** odsyła dane do klienta.
3. Funkcja **close()** kończy utworzone zadanie procesu roboczego.

Przykład: tworzenie serwera używającego funkcji spawn(): Przykład ten ilustruje, jak używać funkcji API **spawn()** do tworzenia procesu potomnego, który dziedziczy deskryptor gniazda od procesu macierzystego. Informacje dotyczące wykorzystania przykładowych kodów zawiera sekcja Informacje dotyczące kodu.

```
/**************************************************************************/
/* Aplikacja tworzy za pomocą funkcji spawn() proces potomny.            */
/**************************************************************************/

#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;

   /*************************************************/
   /* Jeśli podano argument, użyj go do sterowania  */
   /* liczbą połączeń przychodzących                */
   /*************************************************/
   if (argc >= 2)
      num = atoi(argv[1]);
   else
      num = 1;

   /*************************************************/
   /* Utwórz gniazdo strumienia AF_INET do          */
   /* odbierania połączeń przychodzących            */
   /*************************************************/
   listen_sd = socket(AF_INET, SOCK_STREAM, 0);
   if (listen_sd < 0)
   {
      perror("Niepowodzenie socket()");
      exit(-1);
   }

   /*************************************************/
   /* Umożliwia ponowne użycie deskryptora gniazda  */
   /*************************************************/
   rc = setsockopt(listen_sd,
                   SOL_SOCKET,  SO_REUSEADDR,
                   (char *)&on, sizeof(on));
   if (rc < 0)
   {
      perror("Niepowodzenie setsockopt()");
      close(listen_sd);
      exit(-1);
   }
```

```
/**************************************************/
/* Powiąż gniazdo                                 */
/**************************************************/
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("Niepowodzenie bind()");
   close(listen_sd);
   exit(-1);
}

/**************************************************/
/* Ustawia nasłuchiwanie parametru backlog.       */
/**************************************************/
rc = listen(listen_sd, 5);
if (rc < 0)
{
   perror("Niepowodzenie listen()");
   close(listen_sd);
   exit(-1);
}

/**************************************************/
/* Poinformuj użytkownika o tym, że               */
/* serwer jest gotowy                             */
/**************************************************/
printf("Serwer jest gotowy\n");

/**************************************************/
/* Przejdź przez pętlę raz dla każdego połączenia*/
/**************************************************/
for (i=0; i < num; i++)
{
   /***********************************************/
   /* Oczekiwanie na połączenie przychodzące     */
   /***********************************************/
   printf("Iteracja: %d\n", i+1);
   printf("  oczekuje na accept()\n");
   accept_sd = accept(listen_sd, NULL, NULL);
   if (accept_sd < 0)
   {
      perror("Niepowodzenie accept()");
      close(listen_sd);
      exit(-1);
   }
   printf(" accept zakończone powodzeniem\n");

   /***********************************************/
   /* Inicjowanie parametrów spawn               */
   /*                                             */

   /* Deskryptor gniazda dla nowego połączenia   */
   /* jest odwzorowywany na deskryptor 0         */
   /* w programie potomnym.                      */
   /***********************************************/
   memset(&inherit, 0, sizeof(inherit));
   spawn_argv[0]  = NULL;
   spawn_envp[0] = NULL;
   spawn_fdmap[0] = accept_sd;

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

```
      /* Utwórz zadanie procesu roboczego          */
      /**********************************************/
      printf("  tworzenie zadania procesu roboczego\n");
      pid = spawn("/QSYS.LIB/QGPL.LIB/WRKR1.PGM",
                  1, spawn_fdmap, &inherit,
                  spawn_argv, spawn_envp);
      if (pid < 0)
      {
         perror("Niepowodzenie spawn()");
         close(listen_sd);
         close(accept_sd);
         exit(-1);
      }
      printf("  spawn zakończona pomyślnie\n");

      /**********************************************/
      /* Zamyka połączenie przychodzące, ponieważ   */
      /* zostało ono przekazane do obsługi przez    */
      /* proces roboczy.                            */
      /**********************************************/
      close(accept_sd);
   }

   /*************************************************/
   /* Zamyka gniazdo nasłuchujące.                  */
   /*************************************************/
   close(listen_sd);
}
```

Sekcja Przykład: umożliwianie przekazania buforu danych do zadania procesu roboczego zawiera przykładowy program, który do kończenia procesów używa deskryptorów gniazd.

Przykład: umożliwianie przekazania buforu danych do zadania procesu roboczego: Przykład ten zawiera kod umożliwiający przekazanie buforu danych z zadania klienta do zadania procesu roboczego i odesłanie go z powrotem. Informacje dotyczące wykorzystania przykładowych kodów zawiera sekcja Informacje dotyczące kodu.

```
/**************************************************************************/
/* Proces roboczy, który odbiera bufor danych i odsyła go do klienta      */
/**************************************************************************/

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

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

   /*************************************************/
   /* Deskryptor dla połączenia przychodzącego jest */
   /* przekazywany do procesu roboczego jako        */
   /* deskryptor 0.                                 */
   /*************************************************/
   sockfd = 0;

   /*************************************************/
   /* Odebranie komunikatu od klienta               */
   /*************************************************/
   printf(" Oczekiwanie na przysłanie komunikatu od klienta\n");
   rc = recv(sockfd, buffer, sizeof(buffer), 0);
   if (rc <= 0)
   {
      perror("Niepowodzenie recv()");
      close(sockfd);
      exit(-1);
```

```
   }
   printf("<%s>\n", buffer);

   /************************************************/
   /* Odesłanie danych do klienta                  */
   /************************************************/
   printf("Zwrot danych\n");
   len = rc;
   rc = send(sockfd, buffer, len, 0);
   if (rc <= 0)
   {
      perror("Niepowodzenie send()");
      close(sockfd);
      exit(-1);
   }

   /************************************************/
   /* Zamknięcie połączenia przychodzącego         */
   /************************************************/
   close(sockfd);

}
```

Przykład: przekazywanie deskryptorów pomiędzy procesami

Przykłady funkcji **sendmsg()** i **recvmsg()** pokazują, jak zaprojektować program serwera, który korzysta z tych funkcji do obsługi połączeń przychodzących. Gdy serwer jest uruchamiany, tworzy pulę zadań procesów roboczych. Te uprzednio przypisane (utworzone) zadania procesów roboczych oczekują na moment, w którym będą potrzebne. Gdy zadanie klienta łączy się z serwerem, serwer przypisuje połączenie przychodzące do jednego z zadań procesów roboczych.

Poniższy rysunek przedstawia współdziałanie serwera, procesu roboczego i klienta, gdy w systemie używany jest program serwera **sendmsg()** i **recvmsg()**.

Uwaga: Opis funkcji wykonywanych przez klienta zawiera sekcja Przykład: ogólny program klienta.

Przebieg zdarzeń w gnieździe: serwer używający funkcji sendmsg() i recvmsg()

Poniżej wyjaśniono sekwencję wywołań funkcji gniazd, przedstawionych na powyższej ilustracji. Opisano także relacje pomiędzy aplikacją serwera a przykładowym procesem roboczym. Każdy zbiór wywołań zawiera odsyłacze do uwag dotyczących użycia poszczególnych funkcji API. Aby uzyskać szczegółowe informacje dotyczące użycia tych funkcji API, należy użyć poniższych odsyłaczy. W sekcji Przykład: program serwera używany dla funkcji sendmsg() i recvmsg() do utworzenia procesu potomnego za pomocą wywołania funkcji **sendmsg()** i **recvmsg()** wykorzystano następujące wywołania funkcji gniazd:

1. Funkcja **socket()** zwraca deskryptor gniazda odpowiadający punktowi końcowemu. Instrukcja ta informuje również, że dla tego gniazda zostanie użyta rodzina adresów INET (Internet Protocol) z transportem TCP transport (SOCK_STREAM).
2. Po utworzeniu deskryptora gniazda funkcja **bind()** pobiera unikalną nazwę gniazda.
3. Funkcja **listen()** umożliwia serwerowi przyjęcie połączenia przychodzącego od klienta.

4. Funkcja **socketpair()** tworzy parę datagramowych gniazdUNIX.Serwer do utworzenia pary gniazd AF_UNIX może użyć funkcji API **socketpair()**.
5. Funkcja **spawn()** inicjuje parametry zadania roboczego do obsługi żądań przychodzących. W tym przykładzie utworzone zadanie potomne dziedziczy deskryptor gniazda utworzony za pomocą funkcji **socketpair()**.
6. Serwer używa funkcji **accept()** do zaakceptowania żądania połączenia przychodzącego. Wywołanie funkcji **accept()** zostanie zablokowane na nieokreślony czas oczekiwania na połączenie przychodzące.
7. Funkcja **sendmsg()** wysyła połączenie przychodzące do jednego z zadań procesu roboczego. Proces potomny akceptuje połączenie przy użyciu funkcji **recvmsg()**. Gdy serwer wywołuje funkcję **sendmsg()**, zadanie potomne nie jest aktywne.
8. W tym przykładzie pierwsze wywołanie funkcji **close()** zamyka zaakceptowane gniazdo. Drugie wywołanie funkcji **close()** kończy gniazdo nasłuchujące.

Przebieg zdarzeń w gnieździe: zadanie procesu roboczego używające funkcji recvmsg()
W sekcji Przykład: program procesu roboczego używany dla funkcji sendmsg() i recvmsg() wykorzystano następujące wywołania funkcji gniazd:

1. Po zaakceptowaniu połączenia przez serwer i przekazaniu deskryptora gniazda do procesu roboczego funkcja **recvmsg()** odbiera deskryptor. W tym przykładzie funkcja **recvmsg()** będzie czekać do momentu, gdy serwer wyśle deskryptor.
2. Funkcja **recv()** odbiera komunikat od klienta.
3. Funkcja **send()** odsyła dane do klienta.
4. Funkcja **close()** kończy zadanie procesu roboczego.

Przykład: program serwera używany dla funkcji sendmsg() i recvmsg(): Przykład ten ilustruje, jak używać funkcji API **sendmsg()** do tworzenia puli zadań procesów roboczych. W sekcji Przykład: ogólny program klienta znajduje się przykład kodu dla typowego zadania klienta, którego można użyć z niniejszym przykładem. Informacje dotyczące wykorzystania przykładowych kodów zawiera sekcja Informacje dotyczące kodu.

```
/**************************************************************************/
/* Serwer używający funkcji sendmsg() do tworzenia procesów roboczych     */
/**************************************************************************/
#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;

   /*************************************************/
   /* Jeśli podano argument, użyj go do sterowania  */
   /* liczbą połączeń przychodzących               */
   /*************************************************/
   if (argc >= 2)
      num = atoi(argv[1]);
   else
      num = 1;

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

```
/* Utwórz gniazdo strumienia AF_INET do          */
/* odbierania połączeń przychodzących            */
/*************************************************/
listen_sd = socket(AF_INET, SOCK_STREAM, 0);
if (listen_sd < 0)
{
   perror("Niepowodzenie socket()");
   exit(-1);
}

/*************************************************/
/* Umożliwia ponowne użycie deskryptora gniazda  */
/*************************************************/
rc = setsockopt(listen_sd,
                SOL_SOCKET,  SO_REUSEADDR,
                (char *)&on, sizeof(on));
if (rc < 0)
{
   perror("Niepowodzenie setsockopt()");
   close(listen_sd);
   exit(-1);
}

/*************************************************/
/* Powiąż gniazdo                                */
/*************************************************/
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("Niepowodzenie bind()");
   close(listen_sd);
   exit(-1);
}

/*************************************************/
/* Ustawia nasłuchiwanie parametru backlog.      */
/*************************************************/
rc = listen(listen_sd, 5);
if (rc < 0)
{
   perror("Niepowodzenie listen()");
   close(listen_sd);
   exit(-1);
}

/*************************************************/
/* Utworzenie pary datagramowych gniazd UNIX     */
/*************************************************/
rc = socketpair(AF_UNIX, SOCK_DGRAM, 0, pair_sd);
if (rc != 0)
{
   perror("Niepowodzenie socketpair()");
   close(listen_sd);
   exit(-1);
}
server_sd = pair_sd[0];
worker_sd = pair_sd[1];

/*************************************************/
/* Inicjowanie parametrów przed wejściem w pętlę */
/*                                               */
/* Deskryptor gniazda proc. rob. jest odwzoro-   */
```

```
/* wywany na deskryptor 0 programu potomnego.    */
/*************************************************/
memset(&inherit, 0, sizeof(inherit));
spawn_argv[0]  = NULL;
spawn_envp[0] = NULL;
spawn_fdmap[0] = worker_sd;

/*************************************************/
/* Utwórz każde z zadań procesów roboczych       */
/*************************************************/
printf("Tworzenie zadań procesów roboczych...\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("Niepowodzenie spawn()");
      close(listen_sd);
      close(server_sd);
      close(worker_sd);
      exit(-1);
   }
   printf("  Proces roboczy = %d\n", pid);
}

/*************************************************/
/* Zamknięcie gniazda procesu roboczego          */
/*************************************************/
close(worker_sd);

/*************************************************/
/* Poinformuj użytkownika o tym, że              */
/* serwer jest gotowy                            */
/*************************************************/
printf("Serwer jest gotowy\n");

/*************************************************/
/* Przejdź przez pętlę raz dla każdego połączenia*/
/*************************************************/
for (i=0; i < num; i++)
{
   /**********************************************/
   /* Oczekiwanie na połączenie przychodzące     */
   /**********************************************/
   printf("Iteracja: %d\n", i+1);
   printf("  oczekuje na accept()\n");
   accept_sd = accept(listen_sd, NULL, NULL);
   if (accept_sd < 0)
   {
      perror("Niepowodzenie accept()");
      close(listen_sd);
      close(server_sd);
      exit(-1);
   }
   printf(" accept zakończone powodzeniem\n");

   /**********************************************/
   /* Zainicjuj strukturę nagłówka komunikatu    */
   /**********************************************/
   memset(&msg,   0, sizeof(msg));

   /**********************************************/
   /* Żadne dane nie są wysyłane, więc nie po-   */
   /* trzeba podawać wartości dla żadnego z pól  */
   /* msg_iov.                                   */
```

```
      /* Wartość memset struktury nagłówka komunika-*/
      /* tu ustawi wskaźnik msg_iov na NULL        */
      /* i w polu msg_iovcnt wartość 0.            */
      /**********************************************/

      /**********************************************/
      /* Jedyne pola struktury nagłówka komunikatu, */
      /* w których trzeba wpisać dane, to pola      */
      /* msg_accrights.                             */
      /**********************************************/
      msg.msg_accrights  = (char *)&accept_sd;
      msg.msg_accrightslen = sizeof(accept_sd);

      /**********************************************/
      /* Przekaż połączenie przychodzące jednemu z  */
      /* procesów roboczych.                        */
      /*                                            */
      /* UWAGA: Nie wiadomo, które zadanie robocze  */
      /* otrzyma połączenie przychodzące.           */
      /**********************************************/
      rc = sendmsg(server_sd, &msg, 0);
      if (rc < 0)
      {
         perror("Niepowodzenie socketpair()");
         close(listen_sd);
         close(accept_sd);
         close(server_sd);
         exit(-1);
      }
      printf("sendmsg zakończona pomyślnie\n");

      /**********************************************/
      /* Zamyka połączenie przychodzące, ponieważ   */
      /* zostało ono przekazane do obsługi przez    */
      /* proces roboczy.                            */
      /**********************************************/
      close(accept_sd);
   }

   /*************************************************/
   /* Zamknięcie gniazd serwera i nasłuchiwania     */
   /*************************************************/
   close(server_sd);
   close(listen_sd);
}
```

Przykład: program procesu roboczego używany dla funkcji sendmsg () i recvmsg (): Przykład ten ilustruje, jak używać zadania klienta funkcji API **recvmsg()** do odbierania zadań procesów roboczych. Informacje dotyczące wykorzystania przykładowych kodów zawiera sekcja Informacje dotyczące kodu.

```
/**************************************************************************/
/* Zadanie robocze używające funkcji recvmsg() do obsługi żądań klientów  */
/**************************************************************************/
#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;

   /*************************************************/
   /* Jeden z deskryptorów gniazda, który jest zwra-*/
```

```
/* cany przez socketpair(), przekazywany jest do */
/* procesu roboczego jako deskryptor 0.         */
/************************************************/
worker_sd = 0;

/************************************************/
/* Zainicjuj strukturę nagłówka komunikatu      */
/************************************************/
memset(&msg,   0, sizeof(msg));
memset(iov,    0, sizeof(iov));

/************************************************/
/* Wywołanie recvmsg() NIE zostanie zablokowane, */
/* dopóki nie zostanie podany bufor o niezerowej */
/* długości                                     */
/************************************************/
iov[0].iov_base = buffer;
iov[0].iov_len  = sizeof(buffer);
msg.msg_iov     = iov;
msg.msg_iovlen  = 1;

/************************************************/
/* Wypełnienie pola msg_accrights, aby można    */
/* było odebrać deskryptor                      */
/************************************************/
msg.msg_accrights    = (char *)&pass_sd;
msg.msg_accrightslen = sizeof(pass_sd);

/************************************************/
/* Czekaj na przysłanie deskryptora             */
/************************************************/
printf("Oczekiwanie na recvmsg\n");
rc = recvmsg(worker_sd, &msg, 0);
if (rc < 0)
{
   perror("Niepowodzenie recvmsg()");
   close(worker_sd);
   exit(-1);
}
else if (msg.msg_accrightslen <= 0)
{
   printf("Deskryptor nie został odebrany\n");
   close(worker_sd);
   exit(-1);
}
else
{
   printf("Otrzymano deskryptor = %d\n", pass_sd);
}

/************************************************/
/* Odebranie komunikatu od klienta              */
/************************************************/
printf(" Oczekiwanie na przysłanie komunikatu od klienta\n");
rc = recv(pass_sd, buffer, sizeof(buffer), 0);
if (rc <= 0)
{
   perror("Niepowodzenie recv()");
   close(worker_sd);
   close(pass_sd);
   exit(-1);
}
printf("<%s>\n", buffer);

/************************************************/
/* Odesłanie danych do klienta                  */
/************************************************/
```

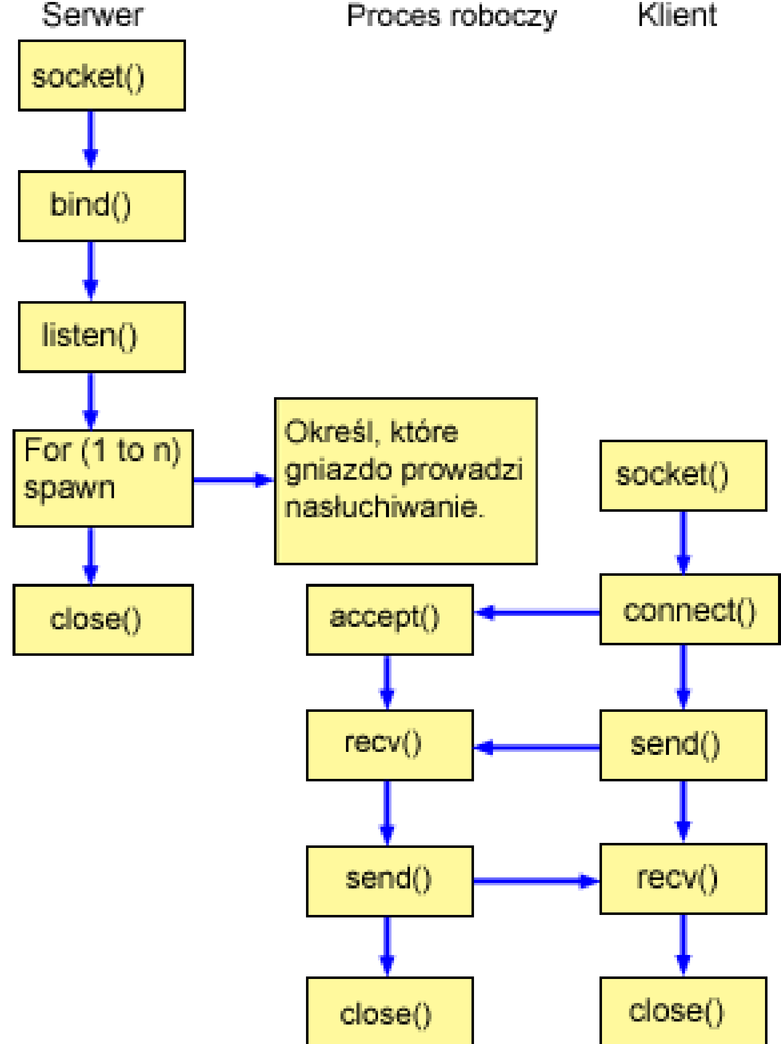
```
   printf("Zwrot danych\n");
   len = rc;
   rc = send(pass_sd, buffer, len, 0);
   if (rc <= 0)
   {
      perror("Niepowodzenie send()");
      close(worker_sd);
      close(pass_sd);
      exit(-1);
   }

   /*************************************************/
   /* Zamknięcie deskryptorów                       */
   /*************************************************/
   close(worker_sd);
   close(pass_sd);
}
```

Przykład: używanie wielu funkcji API accept() do obsługi żądań przychodzących

Przykłady te ilustrują sposób projektowania programu serwera, który używa modelu wielu funkcji **accept()** do obsługi żądań połączeń przychodzących. Gdy uruchamiany jest serwer wielu funkcji **accept()**, uruchamia on funkcje **socket()**, **bind()** i **listen()**. Następnie tworzy pulę zadań procesów roboczych i przypisuje do każdego zadania gniazdo nasłuchujące. Następnie każdy proces roboczy wielu funkcji **accept()** wywołuje funkcję **accept()**.

Poniższy rysunek przedstawia współdziałanie serwera, procesu roboczego i klienta, gdy w systemie używany jest program serwera wielu funkcji **accept()**.

Uwaga: Opis funkcji wykonywanych przez klienta zawiera sekcja Przykład: ogólny program klienta.



Przebieg zdarzeń w gnieździe: serwer tworzący pulę wielu zadań procesów roboczych accept()
Poniżej wyjaśniono sekwencję wywołań funkcji gniazd, przedstawionych na powyższej ilustracji. Opisano także relacje pomiędzy aplikacją serwera a przykładowym procesem roboczym. Każdy zbiór wywołań zawiera odsyłacze do uwag dotyczących użycia poszczególnych funkcji API. Aby uzyskać szczegółowe informacje dotyczące użycia tych funkcji API, należy użyć poniższych odsyłaczy. W sekcji Przykład: program serwera do tworzenia puli wielu procesów roboczych accept() do utworzenia procesu potomnego wykorzystano następujące wywołania funkcji gniazd:

1. Funkcja **socket()** zwraca deskryptor gniazda odpowiadający punktowi końcowemu. Instrukcja ta informuje również, że dla tego gniazda zostanie użyta rodzina adresów INET (Internet Protocol) z transportem TCP transport (SOCK_STREAM).
2. Po utworzeniu deskryptora gniazda funkcja **bind()** pobiera unikalną nazwę gniazda.
3. Funkcja **listen()** umożliwia serwerowi przyjęcie połączenia przychodzącego od klienta.
4. Funkcja **spawn()** tworzy poszczególne zadania procesu roboczego.
5. W tym przykładzie pierwsze wywołanie funkcji **close()** zamyka gniazdo nasłuchujące.

Przebieg zdarzeń w gnieździe: zadanie procesu roboczego używające wielu funkcji accept()
W sekcji Przykład: procesy robocze dla wielu funkcji accept() użyto wywołań następujących funkcji:

1. Po uruchomieniu zadań procesów roboczych przez serwer deskryptor gniazda jest przekazywany do tego zadania w parametrze wiersza komend. Funkcja **accept()** oczekuje na połączenie przychodzące od klienta.
2. Funkcja **recv()** odbiera komunikat od klienta.
3. Funkcja **send()** odsyła dane do klienta.
4. Funkcja **close()** kończy zadanie procesu roboczego.

Przykład: program serwera do tworzenia puli wielu procesów roboczych accept(): Przykład ten ilustruje, jak używać modeli wielu funkcji API **accept()** do tworzenia puli zadań procesów roboczych. W sekcji Przykład: ogólny program klienta znajduje się przykład kodu dla typowego zadania klienta, którego można użyć z niniejszym przykładem. Informacje dotyczące wykorzystania przykładowych kodów zawiera sekcja Informacje dotyczące kodu.

```
/**************************************************************************/
/* Przykładowy program serwera, tworzący pulę zadań roboczych z wieloma   */
/* wywołaniami funkcji 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;

   /*************************************************/
   /* Jeśli podano argument, użyj go do sterowania  */
   /* liczbą połączeń przychodzących                */
   /*************************************************/
   if (argc >= 2)
      num = atoi(argv[1]);
   else
      num = 1;

   /*************************************************/
   /* Utwórz gniazdo strumienia AF_INET do          */
   /* odbierania połączeń przychodzących            */
   /*************************************************/
   listen_sd = socket(AF_INET, SOCK_STREAM, 0);
   if (listen_sd < 0)
   {
      perror("Niepowodzenie socket()");
      exit(-1);
   }

   /*************************************************/
   /* Umożliwia ponowne użycie deskryptora gniazda  */
   /*************************************************/
   rc = setsockopt(listen_sd,
                   SOL_SOCKET,  SO_REUSEADDR,
                   (char *)&on, sizeof(on));
   if (rc < 0)
   {
      perror("Niepowodzenie setsockopt()");
      close(listen_sd);
      exit(-1);
```

```
}

/*************************************************/
/* Powiąż gniazdo                                */
/*************************************************/
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("Niepowodzenie bind()");
   close(listen_sd);
   exit(-1);
}

/*************************************************/
/* Ustawia nasłuchiwanie parametru backlog.      */
/*************************************************/
rc = listen(listen_sd, 5);
if (rc < 0)
{
   perror("Niepowodzenie listen()");
   close(listen_sd);
   exit(-1);
}

/*************************************************/
/* Inicjowanie parametrów przed wejściem w pętlę */
/*                                               */
/* Deskryptor gniazda nasłuchiwania jest odwzoro-*/
/* wywany na deskryptor 0 programu potomnego.    */
/*************************************************/
memset(&inherit, 0, sizeof(inherit));
spawn_argv[0]  = NULL;
spawn_envp[0] = NULL;
spawn_fdmap[0] = listen_sd;

/*************************************************/
/* Utwórz każde z zadań procesów roboczych       */
/*************************************************/
printf("Tworzenie zadań procesów roboczych...\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("Niepowodzenie spawn()");
      close(listen_sd);
      exit(-1);
   }
   printf("  Proces roboczy = %d\n", pid);
}

/*************************************************/
/* Poinformuj użytkownika o tym, że              */
/* serwer jest gotowy                            */
/*************************************************/
printf("Serwer jest gotowy\n");

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

```
   /* Zamknięcie gniazda nasłuchującego            */
   /************************************************/
   close(listen_sd);
}
```

Przykład: procesy robocze dla wielu funkcji accept(): Przykład ten ilustruje użycie wielu funkcji API **accept()** do odebrania zadań procesu roboczego i wywołania serwera **accept()**. Informacje dotyczące wykorzystania przykładowych kodów zawiera sekcja Informacje dotyczące kodu.

```
/**************************************************************************/
/* Zadanie robocze używa wielu wywołań funkcji accept() do obsługi       */
/* połączeń przychodzących od klientów                                    */
/**************************************************************************/
#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];

   /************************************************/
   /* Deskryptor gniazda nasłuchiwania przekazywany */
   /* jest do tego zadania procesu roboczego jako   */
   /* parametr wiersza komend                       */
   /************************************************/
   listen_sd = 0;

   /************************************************/
   /* Oczekiwanie na połączenie przychodzące       */
   /************************************************/
   printf("Oczekiwanie na accept()\n");
   accept_sd = accept(listen_sd, NULL, NULL);
   if (accept_sd < 0)
   {
      perror("Niepowodzenie accept()");
      close(listen_sd);
      exit(-1);
   }
   printf("  Zaakceptowane\n");

   /************************************************/
   /* Odebranie komunikatu od klienta              */
   /************************************************/
   printf(" Oczekiwanie na przysłanie komunikatu od klienta\n");
   rc = recv(accept_sd, buffer, sizeof(buffer), 0);
   if (rc <= 0)
   {
      perror("Niepowodzenie recv()");
      close(listen_sd);
      close(accept_sd);
      exit(-1);
   }
   printf("<%s>\n", buffer);

   /************************************************/
   /* Odesłanie danych do klienta                  */
   /************************************************/
   printf("Zwrot danych\n");
   len = rc;
   rc = send(accept_sd, buffer, len, 0);
   if (rc <= 0)
   {
      perror("Niepowodzenie send()");
      close(listen_sd);
```

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

   /**********************************************/
   /* Zamknięcie deskryptorów                    */
   /**********************************************/
   close(listen_sd);
   close(accept_sd);
}
```

Przykład: ogólny program klienta

Poniższy kod zawiera przykłady podstawowych zadań klienta. Zadanie klienta używa funkcji **socket()**, **connect()** , **send()**, **recv()** i **close()**. Zadanie klienta nie jest powiadamiane, że wysłany bufor danych został odebrany i przekazany do zadania procesu roboczego, a nie do serwera. Aby napisać aplikację klienta, która działa niezależnie od tego, czy serwer używa rodziny adresów AF_INET czy AF_INET6, należy zastosować Przykład: klient IPv4 lub IPv6.

To zadanie klienta będzie działało z każdą z poniższych, zorientowanych na połączenie konfiguracji serwera:

- Serwer iteracyjny. Przykład programu znajduje się w sekcji Przykład: pisanie programu serwera iteracyjnego.
- Potomny serwer i proces roboczy. Przykład programu znajduje się w sekcji Przykład: używanie funkcji API spawn() do tworzenia procesów potomnych.
- Serwer sendmsg() i proces roboczy rcvmsg(). Przykład programu znajduje się w sekcji Przykład: program serwera używany dla funkcji sendmsg() i recvmsg().
- Konstrukcja z wielokrotną funkcją accept(). Przykład programu znajduje się w sekcji Przykład: program serwera do tworzenia puli wielu zadań accept() procesów roboczych.
- Konstrukcja z nieblokującymi operacjami we/wy i funkcją select(). Przykładowy program znajduje się w sekcji Przykład: nieblokujące operacje we/wy i funkcja select().
- Serwer akceptujący połączenia od klientów IPv4 i IPv6. Przykład programu znajduje się w sekcji Przykład: akceptowanie połączeń od klientów IPv6 i IPv4.

Przebieg zdarzeń w gnieździe: ogólny program klienta

Poniższy program używa wywołań następujących funkcji:

1. Funkcja **socket()** zwraca deskryptor gniazda odpowiadający punktowi końcowemu. Instrukcja ta informuje również, że dla tego gniazda zostanie użyta rodzina adresów INET (Internet Protocol) z transportem TCP transport (SOCK_STREAM).
2. Po odebraniu deskryptora gniazda należy użyć funkcji **connect()** do nawiązania połączenia z serwerem.
3. Funkcja **send()** wysyła bufor danych do zadania roboczego (lub zadań roboczych).
4. Funkcja **recv()** odbiera bufor danych z zadania roboczego (lub zadań roboczych).
5. Funkcja **close()** zamyka wszystkie otwarte deskryptory gniazd.

Informacje dotyczące wykorzystania przykładowych kodów zawiera sekcja Informacje dotyczące kodu.

```
/**************************************************************************/
/* Przykład ogólnego klienta dla serwerów zorientowanych na połączenie    */
/**************************************************************************/
#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;

/*************************************************/
/* Utwórz gniazdo strumienia AF_INET             */
/*************************************************/
sockfd = socket(AF_INET, SOCK_STREAM, 0);
if (sockfd < 0)
{
   perror("socket");
   exit(-1);
}

/*************************************************/
/* Inicjowanie struktury adresów gniazd          */
/*************************************************/
memset(&addr, 0, sizeof(addr));
addr.sin_family = AF_INET;
addr.sin_addr.s_addr = htonl(INADDR_ANY);
addr.sin_port   = htons(SERVER_PORT);

/*************************************************/
/* Połączenie z serwerem                         */
/*************************************************/
rc = connect(sockfd,
             (struct sockaddr *)&addr,
             sizeof(struct sockaddr_in));
if (rc < 0)
{
   perror("connect");
   close(sockfd);
   exit(-1);
}
printf("Połączenie nawiązane.\n");

/*************************************************/
/* Wpisanie do buforu danych do wysłania         */
/*************************************************/
printf("Wpisz komunikat do wysłania:\n");
gets(send_buf);

/*************************************************/
/* Wysłanie buforu danych do zadania procesu rob.*/
/*************************************************/
len = send(sockfd, send_buf, strlen(send_buf) + 1, 0);
if (len != strlen(send_buf) + 1)
{
   perror("send");
   close(sockfd);
   exit(-1);
}
printf("Wysłano bajtów: %d\n", len);

/*************************************************/
/* Pobranie buforu danych z zadania procesu rob. */
/*************************************************/
len = recv(sockfd, recv_buf, sizeof(recv_buf), 0);
if (len != strlen(send_buf) + 1)
{
   perror("recv");
   close(sockfd);
   exit(-1);
}
printf("Otrzymano bajtów: %d\n", len);

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

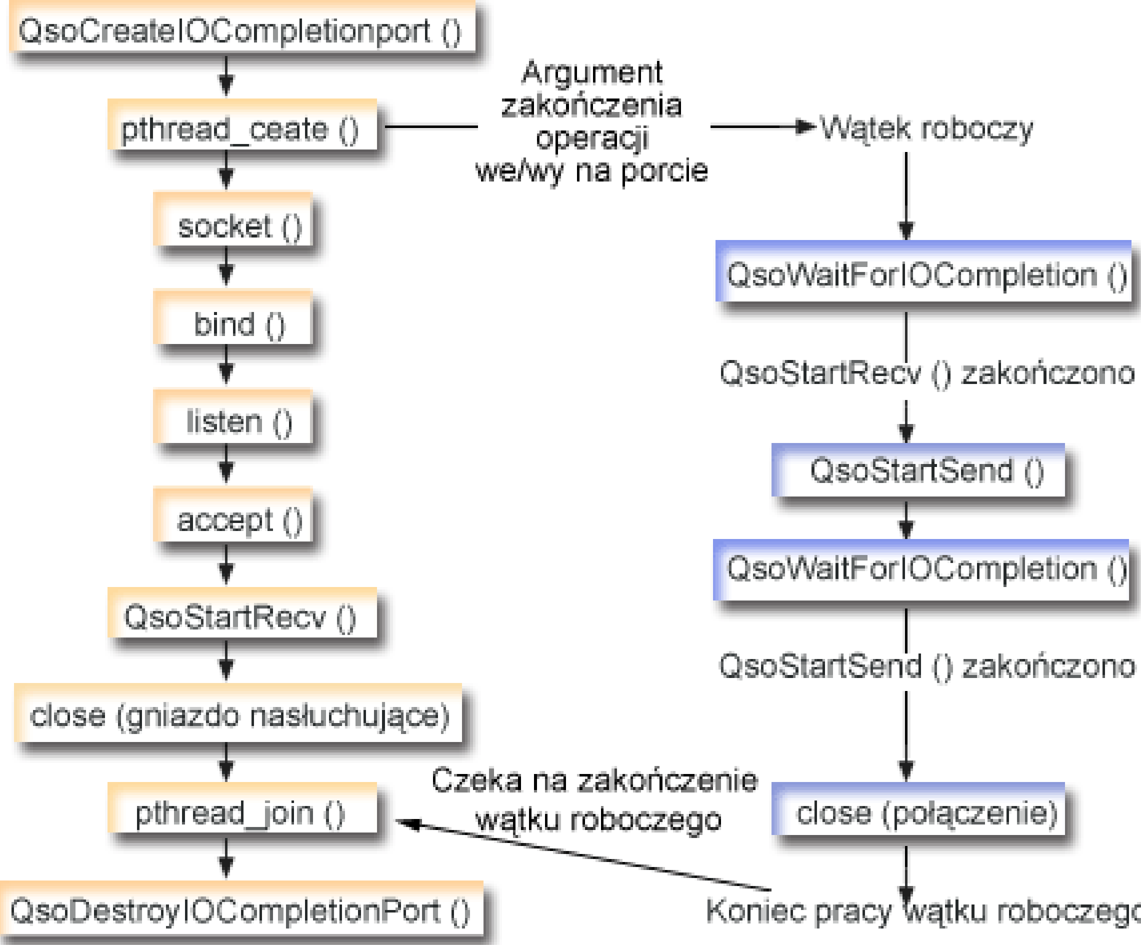
```
   /* Zamknij gniazdo                              */
   /***********************************************/
   close(sockfd);
}
```

Przykład: korzystanie z asynchronicznych operacji we/wy

Aplikacja tworzy port we/wy zakończenia za pomocą funkcji API **QsoCreateIOCompletionPort()**. Funkcja ta zwraca uchwyt, którego można użyć do zaplanowania i oczekiwania na zakończenie żądań asynchronicznych operacji we/wy. Następnie aplikacja uruchamia funkcję operacji wejścia lub operacji wyjścia, podając uchwyt portu we/wy zakończenia. Kiedy operacje we/wy zostają zakończone, informacje o statusie i zdefiniowany przez aplikację uchwyt zostają zapisane w podanym porcie we/wy zakończenia. Zapisanie do portu we/wy zakończenia uaktywnia dokładnie jeden z możliwych wielu wątków oczekujących. Aplikacja odbiera:

- bufor dostarczony w pierwotnym żądaniu,
- długość danych przetworzonych z lub do tego buforu,
- wskazanie typu zakończonej operacji we/wy,
- zdefiniowany przez aplikację uchwyt, który został przekazany w początkowym żądaniu operacji we/wy.

Ten uchwyt aplikacji może być po prostu deskryptorem gniazda, wskazującym połączenie klienta, lub wskaźnikiem do pamięci, która zawiera dodatkowe informacje o stanie połączenia klienta. Ponieważ operacja zakończyła się, a uchwyt aplikacji został przekazany, wątek procesu roboczego określa następny krok w celu zakończenia połączenia klienta. Wątki procesów roboczych, które przetwarzają te zakończone operacje asynchroniczne, mogą mieć wiele różnych żądań klienta i nie są powiązane tylko z jednym z nich. Ponieważ kopiowanie z i do buforów użytkowników odbywa się asynchronicznie względem procesów serwera, czasy oczekiwania dla żądań klientów skracają się. Cecha ta może być szczególnie korzystna w systemach wieloprocesorowych.



Przebieg zdarzeń w gnieździe: asynchroniczny serwer we/wy
Poniżej wyjaśniono sekwencję wywołań funkcji gniazd, przedstawionych na powyższej ilustracji. Opisano także relacje pomiędzy aplikacją serwera a przykładowym procesem roboczym. Każdy zbiór wywołań zawiera odsyłacze do uwag dotyczących użycia poszczególnych funkcji API. Aby uzyskać szczegółowe informacje dotyczące użycia tych funkcji API, należy użyć poniższych odsyłaczy. Diagram ten przedstawia wywołania funkcji gniazd w poniższej przykładowej aplikacji. Serwer ten może współpracować z przykładowym klientem ogólnym.

1. Wątek główny tworzy port we/wy zakończenia, wywołując funkcję **QsoCreateIOCompletionPort()**.
2. Wątek główny tworzy pulę wątków procesów roboczych do przetwarzania wszelkich żądań zakończenia we/wy z portów za pomocą funkcji **pthread_create**.
3. Wątek procesów roboczych wywołuje funkcję **QsoWaitForIOCompletionPort()**, która czeka na żądanie klienta, aby je przetworzyć.
4. Wątek główny akceptuje połączenie klienta i wywołuje funkcję **QsoStartRecv()**, która określa port we/wy zakończenia, pod którym czekają wątki procesów roboczych.

   Uwaga: Akceptacja może również odbyć się w trybie asynchronicznym z użyciem funkcji **QsoStartAccept()**.
5. W pewnym momencie żądanie klienta dociera asynchronicznie do procesu serwera. System operacyjny gniazd wczytuje dostarczony bufor użytkownika i wysyła zakończone żądanie **QsoStartRecv()** do określonego portu we/wy zakończenia. Jeden wątek procesu roboczego zostaje uaktywniony i kontynuuje przetwarzanie tego żądania.
6. Wątek procesu roboczego wyodrębnia deskryptor gniazda z uchwytu zdefiniowanego przez aplikację i odsyła odebrane dane z powrotem do klienta, wykonując operację **QsoStartSend()**.

7. Jeśli dane mogą być wysłane natychmiast, funkcja **QsoStartSend()** zwraca odpowiednią informację, w przeciwnym razie system operacyjny gniazd prześle te dane tak szybko, jak to będzie możliwe i zapisze informację o tym fakcie w określonym porcie we/wy zakończenia. Wątek procesu roboczego pobiera informację o wysłanych danych i czeka w porcie we/wy zakończenia na kolejne żądanie lub zostaje zakończony, jeśli wystąpi taka instrukcja. Do zapisania zdarzenia zakończenia wątku procesu roboczego wątek główny może użyć funkcji **QsoPostIOCompletion()**.
8. Wątek główny czeka na zakończenie zadania przez wątek procesu roboczego, a następnie niszczy port we/wy zakończenia, wywołując funkcję **QsoDestroyIOCompletionPort()**.

Uwaga: Te przykłady programu serwera działają ze standardowym kodem klienta opisanym w sekcji Przykład: ogólny program klienta.

Niniejsze przykłady pokazują, w jaki sposób program serwera może używać funkcji API dla asynchronicznych operacji we/wy. Informacje dotyczące wykorzystania przykładowych kodów zawiera sekcja Informacje dotyczące kodu.

```
      #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);

/********************************************************************/
/*                                                                  */
/* Nazwa funkcji: main                                              */
/*                                                                  */
/* Nazwa opisowa: Wątek główny ustanawia połączenie z klientem oraz */
/* przekazuje przetwarzanie do wątku procesu roboczego.             */
/*                                                                  */
/* Uwaga: Z uwagi na atrybut wątku tego programu należy użyć        */
/* funkcji spawn().                                                 */
/********************************************************************/

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;


       /*********************************************/
       /* Tworzy port we/wy zakończenia dla tego    */
       /* procesu.                                  */
       /*********************************************/
       if ((ioCompPort = QsoCreateIOCompletionPort()) < 0)
       {
         perror("Niepowodzenie QsoCreateIOCompletionPort()");
         exit(-1);
       }
```

```
/*********************************************/
/* Tworzy wątek procesu roboczego w celu     */
/* przetwarzania wszystkich żądań klienta.   */
/* Wątek procesu roboczego będzie oczekiwał  */
/* na żądania klienta napływające do właśnie */
/* utworzonego portu we/wy zakończenia.      */
/*********************************************/
rc = pthread_create(&thr, NULL, workerThread,
                   &ioCompPort);
if (rc < 0)
{
  perror("Niepowodzenie pthread_create()");
  QsoDestroyIOCompletionPort(ioCompPort);
  close(listen_sd);
  exit(-1);
}

/*********************************************/
/* Tworzy gniazdo strumienia AF_INET do      */
/* odbierania połączeń przychodzących        */
/*********************************************/
if ((listen_sd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
{
  perror("Niepowodzenie socket()");
  QsoDestroyIOCompletionPort(ioCompPort);
  exit(-1);
}

/*********************************************/
/*Umożliwia ponowne użycie deskrypt.gniazda  */
/*********************************************/
if ((rc = setsockopt(listen_sd, SOL_SOCKET,
                     SO_REUSEADDR,
                     (char *)&on,
                     sizeof(on))) < 0)
{
  perror("Niepowodzenie setsockopt()");
  QsoDestroyIOCompletionPort(ioCompPort);
  close(listen_sd);
  exit(-1);
}

/*********************************************/
/* Powiąż gniazdo                            */
/*********************************************/
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("Niepowodzenie bind()");
  QsoDestroyIOCompletionPort(ioCompPort);
  close(listen_sd);
  exit(-1);
}

/*********************************************/
/* Ustawia kolejkę (backlog) nasłuchiwania.  */
/*********************************************/
if ((rc = listen(listen_sd, 10)) < 0)
{
  perror("Niepowodzenie listen()");
  QsoDestroyIOCompletionPort(ioCompPort);
```

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

printf("Oczekiwanie na połączenie klienta.\n");

/*********************************************/
/* Akceptuje przychodzące połączenie klienta.*/
/*********************************************/
if ((client_sd = accept(listen_sd, (struct sockaddr *)NULL,
                  NULL)) < 0)
{
  perror("Niepowodzenie accept()");
  QsoDestroyIOCompletionPort(ioCompPort);
  close(listen_sd);
  exit(-1);
}

/*********************************************/
/* Wywołuje QsoStartRecv(), aby odebrać      */
/*            żądanie klienta.               */
/* Uwaga:                                    */
/* postFlag == podczas odczytywania żądania  */
/*            należy je przesłać do portu    */
/*            we/wy zakończenia, nawet       */
/*            wtedy, gdy żądanie jest        */
/*            dostępne natychmiast. Wątek    */
/*            procesu roboczego obsłuży      */
/*            żądanie klienta.               */
/*********************************************/

/*********************************************/
/* Inicjuje strukturę Qso_OverlappedIO_t -   */
/* w polach zastrzeżonych muszą być          */
/* szesnastkowe 00.                          */
/*********************************************/
memset(&ioStruct, '\0', sizeof(ioStruct));

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


/*********************************************/
/* Przechowuje deskryptor klienta w polu     */
/* descriptorHandle Qso_OverlappedIO_t.      */
/* Obszar ten jest używany do przechowywania */
/* informacji określających stan połączenia  */
/* klienta. Pole descriptorHandle jest       */
/* definiowane jako (void *), aby serwer mógł*/
/* w razie potrzeby obsłużyć jak najszerszy  */
/* zakres stanów połączenia klienta.         */
/*********************************************/
*((int*)&ioStruct.descriptorHandle) = client_sd;
ioStruct.postFlag = 1;
ioStruct.fillBuffer = 0;

rc = QsoStartRecv(client_sd, ioCompPort, &ioStruct);
if (rc == -1)
{
  perror("Niepowodzenie QsoStartRecv()");
  QsoDestroyIOCompletionPort(ioCompPort);
  close(listen_sd);
  close(client_sd);
  exit(-1);
}
/*********************************************/
/* Zamknięcie gniazda nasłuchującego serwera */
```

```
       /**********************************************/
       close(listen_sd);

       /**********************************************/
       /* Czeka, aż wątek procesu roboczego zakończy*/
       /* przetwarzanie połączenia klienta.        */
       /**********************************************/
       rc = pthread_join(thr, &status);

       QsoDestroyIOCompletionPort(ioCompPort);
       if ( rc == 0 && (rc = __INT(status)) == Success)
       {
         printf("Sukces.\n");
         exit(0);
       }
       else
       {
         perror("Zgłoszone niepowodzenie pthread_join()");
         exit(-1);
       }
}
/* koniec wątku workerThread */



/********************************************************************/
/*                                                                  */
/* Nazwa funkcji: workerThread                                      */
/*                                                                  */
/* Nazwa opisowa: Przetwarzanie połączenia klienta.                 */
/********************************************************************/
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;

      /**********************************************/
      /* Port we/wy zakończenia jest przekazywany  */
      /* do tej procedury.                         */
      /**********************************************/
      ioCompPort = *(int *)arg;

      /**********************************************/
      /* Czeka przy dostarczonym porcie we/wy      */
      /* zakończenia na żądanie klienta.           */
      /**********************************************/
      waitTime.tv_sec = 500;
      waitTime.tv_usec = 0;
      rc = QsoWaitForIOCompletion(ioCompPort, &ioStruct, &waitTime);
      if (rc == 1 && ioStruct.returnValue != -1)
      /**********************************************/
      /* Odebrano żądanie klienta.                 */
      /**********************************************/
        ;
      else
      {
        printf("Niepowodzenie QsoWaitForIOCompletion() lub QsoStartRecv().\n");
        perror("Niepowodzenie QsoWaitForIOCompletion() lub QsoStartRecv()");
        return __VOID(Failure);
      }
```

```
        /*********************************************/
        /* Uzyskuje deskryptor gniazda powiązanego z */
        /* połączeniem klienta.                      */
        /*********************************************/
        clientfd = *((int *) &ioStruct.descriptorHandle);

        /*********************************************/
        /* Odsyła dane z powrotem do klienta.        */
        /* Uwaga: postFlag == 0. Jeśli zapis zakończy*/
        /* się natychmiast, zostanie zwrócone        */
        /* wskazanie, w przeciwnym razie po dokonaniu*/
        /* zapisu port we/wy zakończenia zostanie    */
        /* zaktualizowany.                           */
        /*********************************************/
        ioStruct.postFlag = 0;
        ioStruct.bufferLength = ioStruct.returnValue;
        rc = QsoStartSend(clientfd, ioCompPort, &ioStruct);

        if (rc == 0)
        /*********************************************/
        /* Operacja zakończona, dane zostały wysłane */
        /*********************************************/
          ;
        else
        {
        /*********************************************/
        /* Dwie możliwości:                          */
        /* rc == -1                                  */
        /* Błąd wywołania funkcji                    */
        /* rc == 1                                   */
        /* Zapis nie mógł odbyć się natychmiast.     */
        /* Po dokonaniu zapisu port we/wy zakończenia*/
        /* zostanie zaktualizowany.                  */
        /*********************************************/

          if (rc == -1)
          {
            printf("Niepowodzenie QsoStartSend().\n");
            perror("Niepowodzenie QsoStartSend()");
            close(clientfd);
            return __VOID(Failure);
          }
         /*********************************************/
         /* Czeka na zakończenie operacji.            */
         /*********************************************/
          rc = QsoWaitForIOCompletion(ioCompPort, &ioStruct, &waitTime);
          if (rc == 1 && ioStruct.returnValue != -1)
         /*********************************************/
         /* Wysłanie zakończone pomyślnie.            */
         /*********************************************/
            ;
          else
          {
            printf("Niepowodzenie QsoWaitForIOCompletion() lub QsoStartSend().\n");
            perror("Niepowodzenie QsoWaitForIOCompletion() lub QsoStartSend()");
            return __VOID(Failure);
          }
        }
        close(clientfd);
        return __VOID(Success);
} /* end workerThread */
```

Przykłady: nawiązywanie chronionych połączeń

Chroniony serwer i klienta można utworzyć przy użyciu funkcji API Global Secure ToolKit (GSKit) lub SSL_. Zalecane jest użycie funkcji API GSKit, ponieważ umożliwiają one nawiązywanie chronionych połączeń przez wszystkie platformy IBM @server. Funkcje API SSL_ są rodzime tylko w systemie OS/400. Oba zestawy funkcji API

gniazd chronionych mają kody powrotu, które pomagają zidentyfikować błędy powstałe podczas nawiązywania połączeń chronionych. Szczegóły dotyczące komunikatów o błędach zawiera sekcja Komunikaty o kodach błędów funkcji API gniazd chronionych.

W poniższych przykładach objaśniono sposób tworzenia chronionego serwera i klienta za pomocą obu metod.
- Przykład: chroniony serwer z asynchronicznym odbieraniem danych
- Przykład: chroniony serwer GSKit z asynchronicznym uzgadnianiem
- Przykład: chroniony klient używający funkcji API GSKit
- Przykład: chroniony serwer używający funkcji API SSL_
- Przykład: chroniony klient używający funkcji API SSL_

Przykład: chroniony serwer GSKit z asynchronicznym odbieraniem danych

Poniższego programu przykładowego można użyć do tworzenia chronionego serwera przy użyciu funkcji API Global Secure ToolKit (GSKit). Serwer otwiera gniazdo, przygotowuje chronione środowisko, akceptuje i przetwarza żądania połączenia, wymienia dane z klientem i kończy sesję. Klient również otwiera gniazdo, ustanawia chronione środowisko, nawiązuje połączenie z serwerem i żąda połączenia chronionego, wymienia dane z serwerem i kończy sesję. Poniższy diagram i towarzyszący mu opis ilustrują przebieg zdarzeń pomiędzy serwerem a klientem.

Uwaga: Poniższe przykłady korzystają z rodziny adresów AF_INET, ale można je zmodyfikować tak, aby używały również rodziny adresów AF_INET6.

Przebieg zdarzeń w gnieździe: chroniony serwer używający asynchronicznego odbierania danych

Serwer
QsoCreateIOCompletionPort ()
pthread_create ()
Wątek roboczy
gsk_environment_open ()
QsoWaitForIOCompletion ()
gsk_attribute_set_enum ()
gsk_secure_soc_startRecv () kończy działanie
gsk_attribute_set_buffer ()
gsk_attribute_get_numeric_value ()
gsk_environment_init ()
gsk_secure_soc_write ()
do klienta
socket ()
bind ()
listen ()
accept ()
gsk_secure_soc_read ()
od klienta
połączenie klienta
Koniec wątku roboczego
gsk_secure_soc_open ()
gsk_attribute_set_numeric_value ()
gsk_secure_soc_init ()
klient gsk_secure_soc_init ()
gsk_secure_soc_startRecv ()
pthread_join ()
czeka na zakończenie wątku roboczego
gsk_secure_soc_close ()
gsk_environment_close ()
close (gniazdo nasłuchujące)
close (zaakceptowane gniazdo)
QsoDestroyIOCompletionPort ()

Część diagramu obejmująca klienta znajduje się w sekcji opisującej chronionego klienta GSKit.

Poniżej wyjaśniono sekwencję wywołań funkcji gniazd, przedstawionych na powyższej ilustracji. Opisano także relacje pomiędzy aplikacją serwera a aplikacją klienta. Każdy zbiór wywołań zawiera odsyłacze do uwag dotyczących użycia poszczególnych funkcji API. Aby uzyskać szczegółowe informacje dotyczące użycia tych funkcji API, należy użyć poniższych odsyłaczy. Diagram ten przedstawia wywołania funkcji gniazd w poniższej przykładowej aplikacji. .

1. Funkcja **QsoCreateIOCompletionPort()** tworzy port we/wy zakończenia.
2. Funkcja **pthread_create** tworzy wątek procesu roboczego, który będzie odbierał dane i odsyłał je do klienta. Wątek procesu roboczego będzie oczekiwał na nadejście żądań od klienta na utworzonym właśnie porcie we/wy zakończenia.
3. Wywołanie funkcji **gsk_environment_open()** w celu uzyskania uchwytu środowiska SSL.
4. Przynajmniej jedno wywołanie funkcji gsk_attribute_set_xxxxx() w celu ustawienia atrybutów środowiska SSL. Minimum to wywołanie funkcji **gsk_attribute_set_buffer()** w celu ustawienia wartości GSK_OS400_APPLICATION_ID lub wartości GSK_KEYRING_FILE. Należy ustawić tylko jedną z tych wartości. Zalecane jest użycie wartości GSK_OS400_APPLICATION_ID. Ponadto należy ustawić typ aplikacji (klient lub serwer), GSK_SESSION_TYPE, używając funkcji **gsk_attribute_set_enum()**.
5. Wywołanie funkcji **gsk_environment_init()** w celu zainicjowania tego środowiska do przetwarzania SSL i określenia informacji o ochronie dla wszystkich sesji SSL, które będą uruchamiane w tym środowisku.
6. Funkcja **socket** tworzy deskryptor gniazda. Następnie serwer wywołuje standardowy zestaw funkcji gniazd: **bind()**, **listen()** i **accept()**, aby możliwe było akceptowanie przychodzących żądań połączenia.
7. Funkcja **gsk_secure_soc_open()** uzyskuje pamięć dla bezpiecznej sesji, ustawia domyślne wartości atrybutów i zwraca uchwyt, który musi zostać zapisany i użyty podczas następnych wywołań funkcji związanych z chronioną sesją.
8. Przynajmniej jedno wywołanie funkcji gsk_attribute_set_xxxxx() w celu ustawienia atrybutów bezpiecznej sesji. Minimum to wywołanie funkcji **gsk_attribute_set_numeric_value()** w celu powiązania określonego gniazda z tą bezpieczną sesją.
9. Wywołanie funkcji **gsk_secure_soc_init()** w celu zainicjowania negocjacji parametrów szyfrowania podczas uzgadniania SSL.

   Uwaga: Aby uzgadnianie SSL się powiodło, program serwera musi okazać certyfikat. Ponadto serwer musi mieć dostęp do klucza prywatnego powiązanego z certyfikatem serwera i zbioru bazy danych kluczy, w którym przechowywany jest certyfikat. W niektórych przypadkach klient także musi okazać certyfikat podczas przetwarzania uzgadniania SSL. Dotyczy to sytuacji, w której serwer, z którym łączy się klient, ma włączone uwierzytelnianie klienta. Wywołania funkcji API **gsk_attribute_set_buffer**(GSK_OS400_APPLICATION_ID) lub **gsk_attribute_set_buffer**(GSK_KEYRING_FILE) identyfikują (różnymi metodami) zbiór bazy danych kluczy, z którego uzyskano klucz prywatny, i certyfikat użyte podczas uzgadniania.
10. Funkcja **gsk_secure_soc_startRecv()** Inicjuje asynchroniczną operację odbioru podczas bezpiecznej sesji.
11. Funkcja **pthread_join** synchronizuje programy serwera i procesu roboczego. Oczekuje ona na zakończenie wątku, odłącza go, a następnie zwraca status wyjścia wątku do serwera.
12. Funkcja **gsk_secure_soc_close()** kończy sesję chronioną.
13. Funkcja **gsk_environment_close()** zamyka środowisko SSL.
14. Funkcja **close()** kończy gniazdo nasłuchujące.
15. Funkcja **close()** kończy gniazdo, które przyjęło połączenie od klienta.
16. Funkcja **QsoDestroyIOCompletionPort()** niszczy port zakończenia.

Przebieg zdarzeń w gnieździe: wątek procesu roboczego używający funkcji API GSKit

1. Utworzony przez aplikację serwera wątek procesu roboczego oczekuje na wysłane przez serwer przychodzące żądanie klienta, aby obsłużyć dane klienta za pomocą wywołania funkcji **gsk_secure_soc_startRecv()**. Funkcja **QsoWaitForIOCompletionPort()** będzie oczekiwać na przekazanym jej przez serwer porcie zakończenia we/wy.

2. Po otrzymaniu żądania klienta funkcja **gsk_attribute_get_numeric_value()** pobiera deskryptor gniazda powiązany z sesją chronioną.
3. Funkcja **gsk_secure_soc_write()** wysyła do klienta komunikat w ramach sesji chronionej.

Informacje dotyczące wykorzystania przykładowych kodów zawiera sekcja Informacje dotyczące kodu.

```
/* Program asynchronicznego serwera GSK używający identyfikatora aplikacji */

/* "IBM udziela niewyłącznej licencji na prawa       */
/* autorskie, stosowanej przy używaniu wszelkich     */
/* przykładowych kodów programów, na podstawie       */
/* których można wygenerować podobne funkcje         */
/* dostosowane do indywidualnych wymagań.            */
/*                                                   */
/* Cały kod przykładowy jest udostępniany przez IBM  */
/* jedynie do celów ilustracyjnych. Programy         */
/* przykładowe nie zostały gruntownie przetestowane. */
/* IBM nie może zatem gwarantować lub sugerować      */
/* niezawodności, użyteczności i funkcjonalności     */
/* tych programów.                                   */
/*                                                   */
/* Wszelkie zawarte tutaj programy są dostarczane    */
/* w stanie, w jakim się znajdują ("AS IS")          */
/* bez udzielania jakichkolwiek gwarancji. Nie udziela*/
/* się domniemanych gwarancji nienaruszania praw osób */
/* trzecich, gwarancji przydatności handlowej        */
/* ani też przydatności do określonego celu."        */

/* Przyjmuje się, że ID aplikacji jest już           */
/* zarejestrowane i powiązane z certyfikatem.        */
/*                                                   */
/* Brak parametrów, trochę komentarzy i wiele wartości*/
/* wpisanych w kodzie, aby przykład był prosty.      */

/* Użyj następującej komendy, aby utworzyć program   */
/* skonsolidowany:                                   */
/* 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);
/********************************************************************/
/* Nazwa opisowa: Wątek główny ustanawia połączenie z klientem oraz */
/* przekazuje przetwarzanie do wątku procesu roboczego.             */
/*                                                                  */
/* Uwaga: Z uwagi na atrybut wątku tego programu należy użyć        */
/* funkcji spawn().                                                 */
/********************************************************************/
int main(void)
{
  gsk_handle my_env_handle=NULL;   /* uchwyt środowiska chronionego */
```

```
gsk_handle my_session_handle=NULL;    /* uchwyt sesji chronionej  */

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;

/*********************************************/
/* Wszystkie komendy są uruchamiane w pętli  */
/* do/while, dzięki czemu czyszczenie odbywa */
/* się na końcu.                             */
/*********************************************/
do
{
  /*********************************************/
  /* Tworzy port we/wy zakończenia dla tego    */
  /* procesu.                                  */
  /*********************************************/
  if ((ioCompPort = QsoCreateIOCompletionPort()) < 0)
  {
    perror("Niepowodzenie QsoCreateIOCompletionPort()");
    break;
  }
  /*********************************************/
  /* Tworzy wątek procesu roboczego w celu     */
  /* przetwarzania wszystkich żądań klienta.   */
  /* Wątek procesu roboczego będzie oczekiwał  */
  /* na żądania klienta napływające do właśnie */
  /* utworzonego portu we/wy zakończenia.      */
  /*********************************************/
  rc = pthread_create(&thr, NULL, workerThread, &ioCompPort);
  if (rc < 0)
  {
    perror("Niepowodzenie pthread_create()");
    break;
  }

  /* otwiera środowisko gsk */
  rc = errno = 0;
  rc = gsk_environment_open(&my_env_handle);
  if (rc != GSK_OK)
  {
    printf("Niepowodzenie gsk_environment_open() z kodem powrotu = %d kod błędu = %d.\n",
           rc,errno);
    printf("kod powrotu %d oznacza %s\n", rc, gsk_strerror(rc));
    break;
  }

  /* ustawia ID aplikacji */
  rc = errno = 0;
  rc = gsk_attribute_set_buffer(my_env_handle,
                              GSK_OS400_APPLICATION_ID,
                              "MY_SERVER_APP",
                              13);
  if (rc != GSK_OK)
  {
    printf("Niepowodzenie gsk_attribute_set_buffer() z kodem powrotu = %d kod błędu = %d.\n"
           ,rc,errno);
    printf("kod powrotu %d oznacza %s\n", rc, gsk_strerror(rc));
    break;
  }

  /* ustawia tę stronę jako serwer          */
```

```
rc = errno = 0;
rc = gsk_attribute_set_enum(my_env_handle,
                            GSK_SESSION_TYPE,
                            GSK_SERVER_SESSION);
if (rc != GSK_OK)
{
  printf("Niepowodzenie gsk_attribute_set_enum() z kodem powrotu = %d kod błędu = %d.\n",
         rc,errno);
  printf("kod powrotu %d oznacza %s\n", rc, gsk_strerror(rc));
  break;
}

/* Protokoły SSL_V2, SSL_V3 i TLS_V1 są włączone domyślnie. */
/* W tym przykładzie wyłączymy protokół SSL_V2.             */
rc = errno = 0;
rc = gsk_attribute_set_enum(my_env_handle,
                            GSK_PROTOCOL_SSLV2,
                            GSK_PROTOCOL_SSLV2_OFF);
if (rc != GSK_OK)
{
  printf("Niepowodzenie gsk_attribute_set_enum() z kodem powrotu = %d kod błędu = %d.\n",
         rc,errno);
  printf("kod powrotu %d oznacza %s\n", rc, gsk_strerror(rc));
  break;
}

/* Określa, jakiego pakietu szyfrującego użyć. Domyślnie włączona jest */
/* domyślna lista szyfrowania. W tym przykładzie użyjemy jednego.      */
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("Niepowodzenie gsk_attribute_set_buffer() z kodem powrotu = %d kod błędu = %d.\n"
         ,rc,errno);
  printf("kod powrotu %d oznacza %s\n", rc, gsk_strerror(rc));
  break;
}

/* Zainicjowanie chronionego środowiska */
rc = errno = 0;
rc = gsk_environment_init(my_env_handle);
if (rc != GSK_OK)
{
  printf("Niepowodzenie gsk_environment_init() z kodem powrotu = %d kod błędu = %d.\n",
         rc,errno);
  printf("kod powrotu %d oznacza %s\n", rc, gsk_strerror(rc));
  break;
}

/* inicjowanie gniazda, które będzie użyte do nasłuchiwania */
lsd = socket(AF_INET, SOCK_STREAM, 0);
if (lsd < 0)
{
  perror("Niepowodzenie socket()");
  break;
}

/* ustawienie gniazda do natychmiastowego ponownego użycia */
rc = setsockopt(lsd, SOL_SOCKET,
                SO_REUSEADDR,
                (char *)&on,
                sizeof(on));
if (rc < 0)
{
```

```
  perror("Niepowodzenie setsockopt()");
  break;
}

/* powiązanie do adresu lokalnego serwera */
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("Niepowodzenie bind()");
  break;
}

/* uaktywnienie gniazda dla przychodzących połączeń klienta */
listen(lsd, 5);
if (rc < 0)
{
  perror("Niepowodzenie listen()");
  break;
}

/* akceptacja przychodzącego połączenia klienta */
al = sizeof(address);
sd = accept(lsd, (struct sockaddr *) &address, &al);
if (sd < 0)
{
  perror("Niepowodzenie accept()");
  break;
}

/* otwarcie sesji chronionej */
rc = errno = 0;
rc = gsk_secure_soc_open(my_env_handle, &my_session_handle);
if (rc != GSK_OK)
{
  printf("Niepowodzenie gsk_secure_soc_open() z kodem powrotu = %d kod błędu = %d.\n",
         rc,errno);
  printf("kod powrotu %d oznacza %s\n", rc, gsk_strerror(rc));
  break;
}

/* powiązanie gniazda z sesją chronioną */
rc=errno=0;
rc = gsk_attribute_set_numeric_value(my_session_handle,
                                     GSK_FD,
                                     sd);
if (rc != GSK_OK)
{
  printf("Niepowodzenie gsk_attribute_set_numeric_value() z kodem powrotu = %d ", rc);
  printf("i numerem błędu = %d.\n", errno);
  printf("kod powrotu %d oznacza %s\n", rc, gsk_strerror(rc));
  break;
}

/* inicjowanie uzgodnienia SSL */
rc = errno = 0;
rc = gsk_secure_soc_init(my_session_handle);
if (rc != GSK_OK)
{
  printf("Niepowodzenie gsk_secure_soc_init() z kodem powrotu = %d kod błędu = %d.\n",
         rc,errno);
  printf("kod powrotu %d oznacza %s\n", rc, gsk_strerror(rc));
  break;
}
```

```
/**********************************************/
/* Wywołanie gsk_secure_soc_startRecv() w    */
/* odebrania żądania od klienta.             */
/* Uwaga:                                    */
/* postFlag == podczas odczytywania żądania  */
/* powinien być zapisany do portu we/wy      */
/* zakończenia, nawet jeśli żądanie jest     */
/* dostępne natychmiast. Wątek roboczy       */
/* przetworzy żądanie klienta.               */
/**********************************************/
/**********************************************/
/* Inicjuje strukturę Qso_OverlappedIO_t -   */
/* w polach zastrzeżonych muszą być          */
/* szesnastkowe 00.                          */
/**********************************************/
memset(&ioStruct, '\0', sizeof(ioStruct));
memset((char *) buff, 0, sizeof(buff));
ioStruct.buffer = buff;
ioStruct.bufferLength = sizeof(buff);

/**********************************************/
/* Przechowuje uchwyt sesji w polu           */
/* descriptorHandle Qso_OverlappedIO_t.      */
/* Obszar ten jest używany do przechowywania */
/* informacji określających stan połączenia  */
/* klienta. Pole descriptorHandle jest       */
/* definiowane jako (void *), aby serwer mógł*/
/* w razie potrzeby obsłużyć jak najszerszy  */
/* zakres stanów połączenia klienta.         */
/**********************************************/
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("Kod powrotu gsk_secure_soc_startRecv() = %d, numer błędu = %d.\n", rc,errno);
  printf("kod powrotu %d oznacza %s\n", rc, gsk_strerror(rc));
  break;
}

/**********************************************/
/* W tym miejscu serwer może wrócić na po-   */
/* czątek pętli, aby przyjąć nowe połączenie.*/
/**********************************************/

/**********************************************/
/* Czeka, aż wątek procesu roboczego zakończy*/
/* przetwarzanie połączenia klienta.         */
/**********************************************/
rc = pthread_join(thr, &status);


/* sprawdzenie statusu procesu roboczego */
if ( rc == 0 && (rc = __INT(status)) == Success)
{
  printf("Sukces.\n");
  successFlag = TRUE;
}
else
{
  perror("Zgłoszone niepowodzenie pthread_join()");
}
```

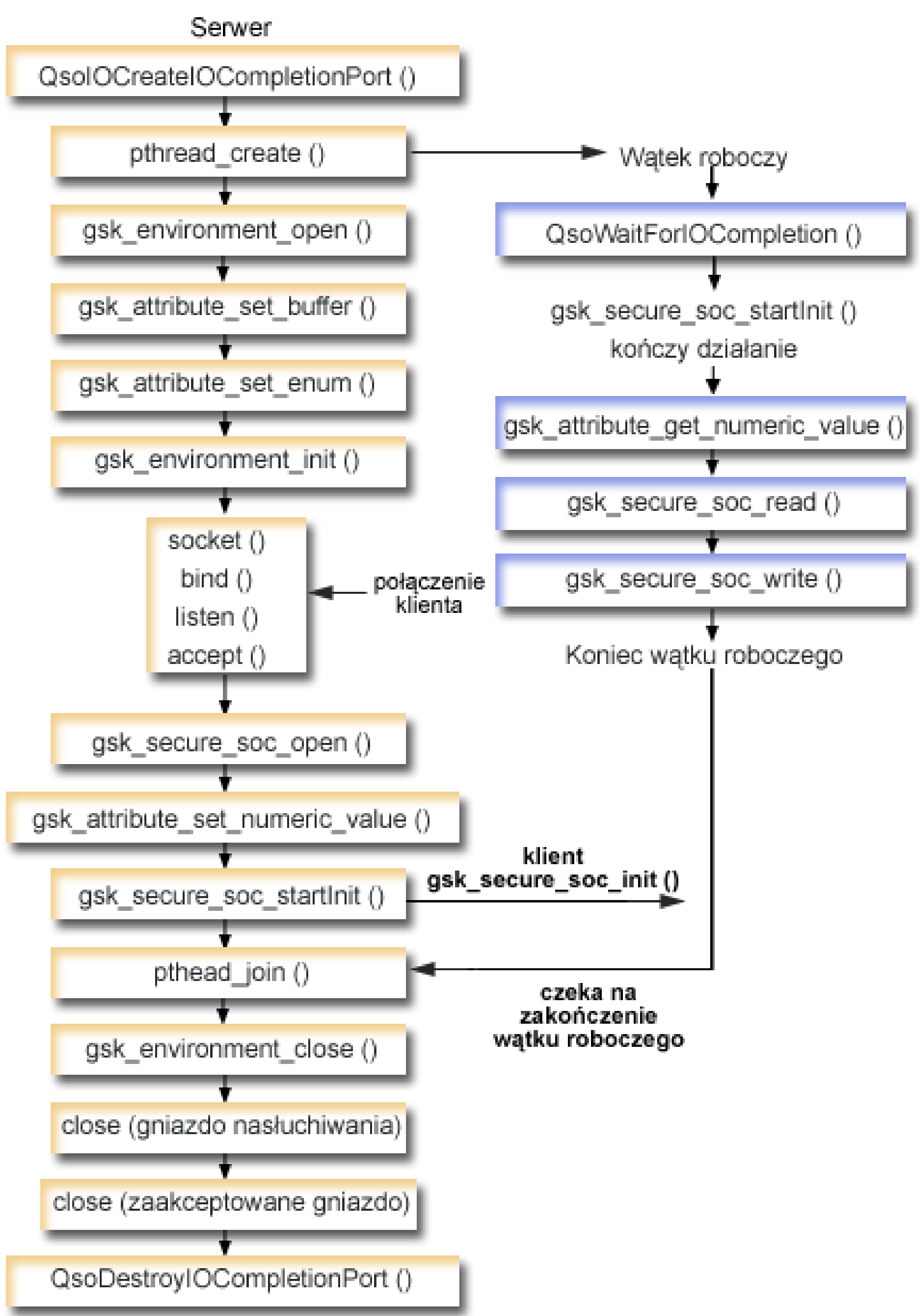
```
  } while(FALSE);

  /* wyłączenie sesji SSL */
  if (my_session_handle != NULL)
    gsk_secure_soc_close(&my_session_handle);

  /* wyłączenie środowiska SSL */
  if (my_env_handle != NULL)
    gsk_environment_close(&my_env_handle);

  /* zamknięcie gniazda nasłuchującego */
  if (lsd > -1)
    close(lsd);
  /* zamknięcie gniazda akceptującego */
  if (sd > -1)
    close(sd);

  /* zniszczenie portu zakończenia */
  if (ioCompPort > -1)
    QsoDestroyIOCompletionPort(ioCompPort);

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


/*******************************************************************/
/* Nazwa funkcji: workerThread                                     */
/*                                                                 */
/* Nazwa opisowa: Przetwarzanie połączenia klienta.                */
/*                                                                 */
/* Uwaga: Aby uprościć przykład, główna procedura obsługuje całe   */
/*        czyszczenie, może ona jednak obsługiwać również uchwyty  */
/*        clientfd i 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;
    /*********************************************/
    /* Port we/wy zakończenia jest przekazywany  */
    /* do tej procedury.                         */
    /*********************************************/
    ioCompPort = *(int *)arg;
    /*********************************************/
    /* Czeka przy dostarczonym porcie we/wy      */
    /* zakończenia na żądanie klienta.           */
    /*********************************************/
    waitTime.tv_sec = 500;
    waitTime.tv_usec = 0;
    rc = QsoWaitForIOCompletion(ioCompPort, &ioStruct, &waitTime);
    if ((rc == 1) &&
        (ioStruct.returnValue == GSK_OK) &&
        (ioStruct.operationCompleted == GSKSECURESOCSTARTRECV))
    /*********************************************/
    /* Odebrano żądanie klienta.                 */
    /*********************************************/
```

```
      ;
    else
    {
      perror("Niepowodzenie QsoWaitForIOCompletion()/gsk_secure_soc_startRecv()");
      printf("ioStruct.returnValue = %d.\n", ioStruct.returnValue);
      return __VOID(Failure);
    }

    /* wyświetlenie wyników na ekranie */
    printf("Funkcja gsk_secure_soc_startRecv() odebrała bajtów: %d, oto one:\n",
           ioStruct.secureDataTransferSize);
    printf("%s\n",ioStruct.buffer);

    /*********************************************/
    /* Uzyskuje uchwyt sesji powiązanej z        */
    /* połączeniem klienta.                      */
    /*********************************************/
    client_session_handle = ioStruct.descriptorHandle;

    /* pobranie gniazda powiązanego z sesją chronioną */
    rc=errno=0;
    rc = gsk_attribute_get_numeric_value(client_session_handle,
                                         GSK_FD,
                                         &clientfd);
    if (rc != GSK_OK)
    {
      printf("Kod powrotu gsk_attribute_get_numeric_value() = %d, numer błędu = %d.\n",
             rc,errno);
      printf("kod powrotu %d oznacza %s\n", rc, gsk_strerror(rc));
      return __VOID(Failure);
    }

    /* wysłanie komunikatu do klienta w ramach bezpiecznej sesji */
    amtWritten = 0;
    rc = gsk_secure_soc_write(client_session_handle,
                              ioStruct.buffer,
                              ioStruct.secureDataTransferSize,
                              &amtWritten);
    if (amtWritten != ioStruct.secureDataTransferSize)
    {
      if (rc != GSK_OK)
      {
        printf("Kod powrotu gsk_secure_soc_write() = %d, numer błędu = %d.\n",
               rc,errno);
        printf("kod powrotu %d oznacza %s\n", rc, gsk_strerror(rc));
        return __VOID(Failure);
      }
      else
      {
        printf("Funkcja gsk_secure_soc_write() nie zapisała wszystkich danych.\n");
        return __VOID(Failure);
      }
    }

    /* wyświetlenie wyników na ekranie */
    printf("Funkcja gsk_secure_soc_write() zapisała bajtów: %d ...\n", amtWritten);
    printf("%s\n",ioStruct.buffer);

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

Przykład: chroniony serwer GSKit z asynchronicznym uzgadnianiem

Od wersji V5R2 systemu OS/400 gniazda obsługują funkcję API **gsk_secure_soc_startInit()**. Funkcja ta umożliwia tworzenie chronionych aplikacji serwera, które mogą obsługiwać żądania w sposób asynchroniczny. Poniższy kod stanowi przykład, jak można użyć tej funkcji. Jest on podobny do przykładu chronionego serwera GSKit z asynchronicznym odbieraniem danych, ale do rozpoczęcia chronionej sesji używa opisanej tu funkcji API.

Poniższa ilustracja przedstawia wywołania funkcji służące do negocjacji uzgadniania asynchronicznego w serwerze chronionym:


Serwer
QsoIOCreateIOCompletionPort ()
pthread_create ()
Wątek roboczy
gsk_environment_open ()
QsoWaitForIOCompletion ()
gsk_attribute_set_buffer ()
gsk_secure_soc_startInit ()
kończy działanie
gsk_attribute_set_enum ()
gsk_attribute_get_numeric_value ()
gsk_environment_init ()
gsk_secure_soc_read ()
socket ()
bind ()
listen ()
accept ()
połączenie
klienta
gsk_secure_soc_write ()
Koniec wątku roboczego
gsk_secure_soc_open ()
gsk_attribute_set_numeric_value ()
klient
gsk_secure_soc_init ()
gsk_secure_soc_startInit ()
pthead_join ()
czeka na
zakończenie
wątku roboczego
gsk_environment_close ()
close (gniazdo nasłuchiwania)
close (zaakceptowane gniazdo)
QsoDestroyIOCompletionPort ()

Część diagramu obejmująca klienta znajduje się w sekcji opisującej klienta GSKit.

Przebieg zdarzeń w gnieździe: chroniony serwer GSKit z asynchronicznym uzgadnianiem

Diagram ten przedstawia wywołania funkcji gniazd w poniższej przykładowej aplikacji.

1. Funkcja **QsoCreateIOCompletionPort()** tworzy port we/wy zakończenia.
2. Funkcja **pthread_create** tworzy wątek procesu roboczego, który będzie przetwarzał wszystkie żądania klienta. Wątek procesu roboczego będzie oczekiwał na nadejście żądań od klienta na utworzonym właśnie porcie we/wy zakończenia.
3. Wywołanie funkcji **gsk_environment_open()** w celu uzyskania uchwytu środowiska SSL.
4. Przynajmniej jedno wywołanie funkcji gsk_attribute_set_xxxxx() w celu ustawienia atrybutów środowiska SSL. Minimum to wywołanie funkcji **gsk_attribute_set_buffer()** w celu ustawienia wartości GSK_OS400_APPLICATION_ID lub wartości GSK_KEYRING_FILE. Należy ustawić tylko jedną z tych wartości. Zalecane jest użycie wartości GSK_OS400_APPLICATION_ID. Ponadto należy ustawić typ aplikacji (klient lub serwer), GSK_SESSION_TYPE, używając funkcji **gsk_attribute_set_enum()**.
5. Wywołanie funkcji **gsk_environment_init()** w celu zainicjowania tego środowiska do przetwarzania SSL i określenia informacji o ochronie dla wszystkich sesji SSL, które będą uruchamiane w tym środowisku.
6. Funkcja **socket** tworzy deskryptor gniazda. Następnie serwer wywołuje standardowy zestaw funkcji gniazd: **bind()**, **listen()** i **accept()**, aby możliwe było akceptowanie przychodzących żądań połączenia.
7. Funkcja **gsk_secure_soc_open()** uzyskuje pamięć dla bezpiecznej sesji, ustawia domyślne wartości atrybutów i zwraca uchwyt, który musi zostać zapisany i użyty podczas następnych wywołań funkcji związanych z chronioną sesją.
8. Przynajmniej jedno wywołanie funkcji gsk_attribute_set_xxxxx() w celu ustawienia atrybutów bezpiecznej sesji. Minimum to wywołanie funkcji **gsk_attribute_set_numeric_value()** w celu powiązania określonego gniazda z tą bezpieczną sesją.
9. Funkcja **gsk_secure_soc_startInit()** uruchamia asynchroniczną negocjację bezpiecznej sesji, używając zestawu atrybutów dla środowiska SSL i bezpiecznej sesji. W tym miejscu sterowanie jest z powrotem przekazywane do programu. Po zakończeniu procesu uzgadniania port zakończenia zostanie zaktualizowany razem z rezultatami. Wątek może kontynuować przetwarzanie; jednakże dla uproszczenia program będzie oczekiwał na zakończenie pracy wątku procesu roboczego.

   Uwaga: Aby uzgadnianie SSL się powiodło, program serwera musi okazać certyfikat. Ponadto serwer musi mieć dostęp do klucza prywatnego powiązanego z certyfikatem serwera i zbioru bazy danych kluczy, w którym przechowywany jest certyfikat. W niektórych przypadkach klient także musi okazać certyfikat podczas przetwarzania uzgadniania SSL. Dotyczy to sytuacji, w której serwer, z którym łączy się klient, ma włączone uwierzytelnianie klienta. Wywołania funkcji API **gsk_attribute_set_buffer** (GSK_OS400_APPLICATION_ID) lub **gsk_attribute_set_buffer** (GSK_KEYRING_FILE) identyfikują (różnymi metodami) zbiór bazy danych kluczy, z którego uzyskano klucz prywatny, i certyfikat użyte podczas uzgadniania.
10. Funkcja **pthread_join** synchronizuje programy serwera i procesu roboczego. Oczekuje ona na zakończenie wątku, odłącza go, a następnie zwraca status wyjścia wątku do serwera.
11. Funkcja **gsk_secure_soc_close()** kończy sesję chronioną.
12. Funkcja **gsk_environment_close()** zamyka środowisko SSL.
13. Funkcja **close()** kończy gniazdo nasłuchujące.
14. Funkcja **close()** kończy gniazdo, które przyjęło połączenie od klienta.
15. Funkcja **QsoDestroyIOCompletionPort()** niszczy port zakończenia.

Przebieg zdarzeń w gnieździe: wątek procesu roboczego przetwarzający chronione żądania asynchroniczne

1. Utworzony przez aplikację serwera wątek procesu roboczego oczekuje na wysłane przez serwer przychodzące żądanie klienta, aby obsłużyć dane klienta. Funkcja **QsoWaitForIOCompletionPort()** będzie oczekiwać na przekazanym jej przez serwer porcie zakończenia we/wy. Wywołanie to oczekuje na zakończenie działania funkcji **gsk_secure_soc_startInit()**.

2. Po otrzymaniu żądania klienta funkcja **gsk_attribute_get_numeric_value()** pobiera deskryptor gniazda powiązany z sesją chronioną.
3. Funkcja **gsk_secure_soc_read()** odbiera od klienta komunikat w ramach sesji chronionej.
4. Funkcja **gsk_secure_soc_write()** wysyła do klienta komunikat w ramach sesji chronionej.

Informacje dotyczące wykorzystania przykładowych kodów zawiera sekcja Informacje dotyczące kodu.

```
/* Program asynchronicznego serwera GSK używający identyfikatora aplikacji */
/* i funkcji gsk_secure_soc_startInit()                                    */

/* Przyjmuje się, że ID aplikacji jest już           */
/* zarejestrowane i powiązane z certyfikatem.        */
/*                                                   */
/* Brak parametrów, trochę komentarzy i wiele wartości*/
/* wpisanych w kodzie, aby przykład był prosty.      */

/* Użyj następującej komendy, aby utworzyć program   */
/* skonsolidowany:                                   */
/* 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);
/********************************************************************/
/* Nazwa opisowa: Wątek główny ustanawia połączenie z klientem oraz */
/* przekazuje przetwarzanie do wątku procesu roboczego.             */
/*                                                                  */
/* Uwaga: Z uwagi na atrybut wątku tego programu należy użyć        */
/* funkcji spawn().                                                 */
/********************************************************************/
int main(void)
{
  gsk_handle my_env_handle=NULL;   /* uchwyt środowiska chronionego */
  gsk_handle my_session_handle=NULL;    /* uchwyt sesji chronionej  */

  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;

  /*********************************************/
  /* Wszystkie komendy są uruchamiane w pętli  */
  /* do/while, dzięki czemu czyszczenie odbywa */
  /* się na końcu.                             */
  /*********************************************/

  do
```

```
{
  /********************************************/
  /* Tworzy port we/wy zakończenia dla tego    */
  /* procesu.                                  */
  /********************************************/
  if ((ioCompPort = QsoCreateIOCompletionPort()) < 0)
  {
    perror("Niepowodzenie QsoCreateIOCompletionPort()");
    break;
  }
  /********************************************/
  /* Tworzy wątek procesu roboczego w celu     */
  /* przetwarzania wszystkich żądań klienta.   */
  /* Wątek procesu roboczego będzie oczekiwał  */
  /* na żądania klienta napływające do właśnie */
  /* utworzonego portu we/wy zakończenia.      */
  /********************************************/
  rc = pthread_create(&thr, NULL, workerThread, &ioCompPort);
  if (rc < 0)
  {
    perror("Niepowodzenie pthread_create()");
    break;
  }

  /* otwiera środowisko gsk */
  rc = errno = 0;
  printf("gsk_environment_open()\n");
  rc = gsk_environment_open(&my_env_handle);
  if (rc != GSK_OK)
  {
  printf("Niepowodzenie gsk_environment_open() z kodem powrotu = %d kod błędu = %d.\n",
           rc,errno);
    printf("kod powrotu %d oznacza %s\n", rc, gsk_strerror(rc));
    break;
  }

  /* ustawia ID aplikacji */
  rc = errno = 0;
  rc = gsk_attribute_set_buffer(my_env_handle,
                                GSK_OS400_APPLICATION_ID,
                                "MY_SERVER_APP",
                                13);
  if (rc != GSK_OK)
  {
    printf("Niepowodzenie gsk_attribute_set_buffer() z kodem powrotu = %d kod błędu = %d.\n"
           ,rc,errno);
    printf("kod powrotu %d oznacza %s\n", rc, gsk_strerror(rc));
    break;
  }

  /* ustawia tę stronę jako serwer          */
  rc = errno = 0;
  rc = gsk_attribute_set_enum(my_env_handle,
                              GSK_SESSION_TYPE,
                              GSK_SERVER_SESSION);
  if (rc != GSK_OK)
  {
    printf("Niepowodzenie gsk_attribute_set_enum() z kodem powrotu = %d kod błędu = %d.\n",
           rc,errno);
    printf("kod powrotu %d oznacza %s\n", rc, gsk_strerror(rc));
    break;
  }

  /* Protokoły SSL_V2, SSL_V3 i TLS_V1 są włączone domyślnie. */
  /* W tym przykładzie wyłączymy protokół SSL_V2.             */
  rc = errno = 0;
  rc = gsk_attribute_set_enum(my_env_handle,
```

```
                         GSK_PROTOCOL_SSLV2,
                            GSK_PROTOCOL_SSLV2_OFF);
if (rc != GSK_OK)
{
  printf("Niepowodzenie gsk_attribute_set_enum() z kodem powrotu = %d kod błędu = %d.\n",
         rc,errno);
  printf("kod powrotu %d oznacza %s\n", rc, gsk_strerror(rc));
  break;
}

/* Określa, jakiego pakietu szyfrującego użyć. Domyślnie włączona jest */
/* domyślna lista szyfrowania. W tym przykładzie użyjemy jednego.      */
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("Niepowodzenie gsk_attribute_set_buffer() z kodem powrotu = %d kod błędu = %d.\n"
         ,rc,errno);
  printf("kod powrotu %d oznacza %s\n", rc, gsk_strerror(rc));
  break;
}

/* Zainicjowanie chronionego środowiska */
rc = errno = 0;
printf("gsk_environment_init()\n");
rc = gsk_environment_init(my_env_handle);
if (rc != GSK_OK)
{
  printf("Niepowodzenie gsk_environment_init() z kodem powrotu = %d kod błędu = %d.\n",
         rc,errno);
  printf("kod powrotu %d oznacza %s\n", rc, gsk_strerror(rc));
  break;
}

/* inicjowanie gniazda, które będzie użyte do nasłuchiwania */
printf("socket()\n");
lsd = socket(AF_INET, SOCK_STREAM, 0);
if (lsd < 0)
{
  perror("Niepowodzenie socket()");
  break;
}

/* ustawienie gniazda do natychmiastowego ponownego użycia */
rc = setsockopt(lsd, SOL_SOCKET,
                SO_REUSEADDR,
                (char *)&on,
                sizeof(on));
if (rc < 0)
{
  perror("Niepowodzenie setsockopt()");
  break;
}

/* powiązanie do adresu lokalnego serwera */
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("Niepowodzenie bind()");
```

```
  break;
}

/* uaktywnienie gniazda dla przychodzących połączeń klienta */
printf("listen()\n");
listen(lsd, 5);
if (rc < 0)
{
  perror("Niepowodzenie listen()");
  break;
}

/* akceptacja przychodzącego połączenia klienta */
al = sizeof(address);
printf("accept()\n");
sd = accept(lsd, (struct sockaddr *) &address, &al);
if (sd < 0)
{
  perror("Niepowodzenie accept()");
  break;
}

/* otwarcie sesji chronionej */
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("Niepowodzenie gsk_secure_soc_open() z kodem powrotu = %d kod błędu = %d.\n",
         rc,errno);
  printf("kod powrotu %d oznacza %s\n", rc, gsk_strerror(rc));
  break;
}
/* powiązanie gniazda z sesją chronioną */
rc=errno=0;
rc = gsk_attribute_set_numeric_value(my_session_handle,
                                     GSK_FD,
                                     sd);
if (rc != GSK_OK)
{
  printf("Niepowodzenie gsk_attribute_set_numeric_value() z kodem powrotu = %d ", rc);
  printf("i numerem błędu = %d.\n", errno);
  printf("kod powrotu %d oznacza %s\n", rc, gsk_strerror(rc));
  break;
}

/********************************************/
/* Wywołanie gsk_secure_soc_startInit() w    */
/* celu asynchronicznego przetworzenia      */
/* uzgadniania SSL                          */
/********************************************/
/********************************************/
/* Inicjuje strukturę Qso_OverlappedIO_t -   */
/* w polach zastrzeżonych muszą być         */
/* szesnastkowe 00.                         */
/********************************************/
memset(&ioStruct, '\0', sizeof(ioStruct));

/********************************************/
/* Przechowuje uchwyt sesji w polu          */
/* descriptorHandle Qso_OverlappedIO_t.     */
/* Obszar ten jest używany do przechowywania */
/* informacji określających stan połączenia  */
/* klienta. Pole descriptorHandle jest      */
/* definiowane jako (void *), aby serwer mógł*/
/* w razie potrzeby obsłużyć jak najszerszy  */
/* zakres stanów połączenia klienta.        */
```

```
    /*********************************************/
    ioStruct.descriptorHandle = my_session_handle;

    /* inicjowanie uzgodnienia SSL */
    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("Kod powrotu gsk_secure_soc_startInit() = %d, numer błędu = %d.\n",rc,errno);
      printf("kod powrotu %d oznacza %s\n", rc, gsk_strerror(rc));
      break;
    }
    else
      printf("Funkcja gsk_secure_soc_startInit odebrała GSK_OS400_ASYNCHRONOUS_SOC_INIT\n");

    /*********************************************/
    /* W tym miejscu serwer może wrócić na po-   */
    /* czątek pętli, aby przyjąć nowe połączenie.*/
    /*********************************************/

    /*********************************************/
    /* Czeka, aż wątek procesu roboczego zakończy*/
    /* przetwarzanie połączenia klienta.         */
    /*********************************************/
    rc = pthread_join(thr, &status);


    /* sprawdzenie statusu procesu roboczego */
    if ( rc == 0 && (rc = __INT(status)) == Success)
    {
      printf("Sukces.\n");
      printf("Sukces.\n");
      successFlag = TRUE;
    }
    else
    {
      perror("Zgłoszone niepowodzenie pthread_join()");
    }
  } while(FALSE);

  /* wyłączenie sesji SSL */
  if (my_session_handle != NULL)
    gsk_secure_soc_close(&my_session_handle);

  /* wyłączenie środowiska SSL */
  if (my_env_handle != NULL)
    gsk_environment_close(&my_env_handle);

  /* zamknięcie gniazda nasłuchującego */
  if (lsd > -1)
    close(lsd);
  /* zamknięcie gniazda akceptującego */
  if (sd > -1)
    close(sd);

  /* zniszczenie portu zakończenia */
  if (ioCompPort > -1)
    QsoDestroyIOCompletionPort(ioCompPort);

  if (successFlag)
    exit(0);

  exit(-1);
}
```

```
/********************************************************************/
/* Nazwa funkcji: workerThread                                      */
/*                                                                  */
/* Nazwa opisowa: Przetwarzanie połączenia klienta.                 */
/*                                                                  */
/* Uwaga: Aby uprościć przykład, główna procedura obsługuje całe    */
/*        czyszczenie, może ona jednak obsługiwać również uchwyty   */
/*        clientfd i 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;
    /*********************************************/
    /* Port we/wy zakończenia jest przekazywany  */
    /* do tej procedury.                         */
    /*********************************************/
    ioCompPort = *(int *)arg;
    /*********************************************/
    /* Czeka przy dostarczonym porcie we/wy      */
    /* na zakończenie uzgadniania SSL.           */
    /*********************************************/
    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))
    /*********************************************/
    /* Uzgadnianie SSL zostało zakończone.       */
    /*********************************************/
    ;
    else
    {
      printf("Niepowodzenie QsoWaitForIOCompletion()/gsk_secure_soc_startInit().\n");
      printf("rc == %d, returnValue - %d, operationCompleted = %d\n",
             rc, ioStruct.returnValue, ioStruct.operationCompleted);
      perror("Niepowodzenie QsoWaitForIOCompletion()/gsk_secure_soc_startInit()");
      return __VOID(Failure);
    }

    /*********************************************/
    /* Uzyskuje uchwyt sesji powiązanej z        */
    /* połączeniem klienta.                      */
    /*********************************************/
    client_session_handle = ioStruct.descriptorHandle;

    /* pobranie gniazda powiązanego z sesją chronioną */
    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("Kod powrotu gsk_attribute_get_numeric_value() = %d, numer błędu = %d.\n",
             rc,errno);
      printf("kod powrotu %d oznacza %s\n", rc, gsk_strerror(rc));
      return __VOID(Failure);
    }
    /* Funkcja memset zeruje bufor szesnastkowo      */
    memset((char *) buff, 0, sizeof(buff));
    amtRead = 0;
    /* odebranie komunikatu od klienta w ramach bezpiecznej sesji */
    printf("gsk_secure_soc_read()\n");
    rc = gsk_secure_soc_read(client_session_handle,
                             buff,
                             sizeof(buff),
                             &amtRead);

    if (rc != GSK_OK)
    {
      printf("Kod powrotu gsk_secure_soc_read() = %d, numer błędu = %d.\n",rc,errno);
      printf("kod powrotu %d oznacza %s\n", rc, gsk_strerror(rc));
      return;
    }

    /* wyświetlenie wyników na ekranie */
    printf("Funkcja gsk_secure_soc_read() odebrała bajtów: %d, oto one:\n",
         amtRead);
    printf("%s\n",buff);


    /* wysłanie komunikatu do klienta w ramach bezpiecznej sesji */
    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("Kod powrotu gsk_secure_soc_write() = %d, numer błędu = %d.\n",rc,errno);
        printf("kod powrotu %d oznacza %s\n", rc, gsk_strerror(rc));
        return __VOID(Failure);
      }
      else
      {
        printf("Funkcja gsk_secure_soc_write() nie zapisała wszystkich danych.\n");
        return __VOID(Failure);
      }
    }
    /* wyświetlenie wyników na ekranie */
    printf("Funkcja gsk_secure_soc_write() zapisała bajtów: %d ...\n", amtWritten);
    printf("%s\n",buff);

    return __VOID(Success);
}
/* koniec wątku workerThread */
```

Przykład: chroniony klient używający funkcji API Global Secure ToolKit (GSKit)

Poniższy kod stanowi przykład klienta używającego funkcji API GSKit. Informacje dotyczące wykorzystania przykładowych kodów zawiera sekcja Informacje dotyczące kodu.

Poniższa ilustracja przedstawia wywołania funkcji w kliencie chronionym używającym funkcji API GSKit:

Klient GSK
gsk_ennrooent_open ()
gsk_attribute_set_buuer ()
gsk_attribute_set_enum ()
gsk_ennrooent__it ()
socket ()
accept ()
connect ()
gsk_secure_soc_open ()
gsk_attribute_set_numeric_value ()
łączy się z
gsk_secure_soc_startinit ()
(asynchroniczne uzgadnianie)
gsk_secure_soc__it ()
łączy się z
gsk_secure_soc_init ()
(asynchroniczne odbieranie danych)
łączy z
gsk_secure_soc_read ()
(asynchroniczne uzgadnianie)
gsk_secure_soc_write ()
łączenie z
gsk_secure_soc_startRecv ()
zakończone
(asynchroniczne odbieranie danych)
gsk_secure_soc_write ()
gsk_secure_soc_read ()
gsk_secure_soc_close ()
gsk_ennrooent_close ()
close

Przebieg zdarzeń w gnieździe: klient GSKit

Diagram ten przedstawia wywołania funkcji gniazd w poniższej przykładowej aplikacji. Klienta tego można używać z przykładowym serwerem GSKit i przykładowym chronionym serwerem GSKit z uzgadnianiem asynchronicznym.

1. Funkcja **gsk_environment_open()** jest wywoływana w celu uzyskania uchwytu środowiska SSL.
2. Przynajmniej jedno wywołanie funkcji gsk_attribute_set_xxxxx() w celu ustawienia atrybutów środowiska SSL. Minimum to wywołanie funkcji **gsk_attribute_set_buffer()** w celu ustawienia wartości GSK_OS400_APPLICATION_ID lub wartości GSK_KEYRING_FILE. Należy ustawić tylko jedną z tych wartości. Zalecane jest użycie wartości GSK_OS400_APPLICATION_ID. Ponadto należy ustawić typ aplikacji (klient lub serwer), GSK_SESSION_TYPE, używając funkcji **gsk_attribute_set_enum()**.
3. Wywołanie funkcji **gsk_environment_init()** w celu zainicjowania tego środowiska do przetwarzania SSL i określenia informacji o ochronie dla wszystkich sesji SSL, które będą uruchamiane w tym środowisku.
4. Funkcja **socket** tworzy deskryptor gniazda. Następnie klient wywołuje funkcję **connect()** w celu połączenia się z aplikacją serwera.
5. Funkcja **gsk_secure_soc_open()** uzyskuje pamięć dla bezpiecznej sesji, ustawia domyślne wartości atrybutów i zwraca uchwyt, który musi zostać zapisany i użyty podczas następnych wywołań funkcji związanych z chronioną sesją.
6. Funkcja **gsk_attribute_set_numeric_value()** przypisuje konkretne gniazdo do tej sesji chronionej.
7. Funkcja **gsk_secure_soc_init()** uruchamia asynchroniczną negocjację bezpiecznej sesji, używając zestawu atrybutów dla środowiska SSL i bezpiecznej sesji.
8. Funkcja **gsk_secure_soc_write()** zapisuje dane w sesji chronionej do wątku procesu roboczego.

   Uwaga: W przykładzie serwera GSKit funkcja ta zapisuje dane do wątku procesu roboczego po zakończeniu działania funkcji **gsk_secure_soc_startRecv()**. W przykładzie serwera asynchronicznego zapisuje ona dane do zakończonej funkcji **gsk_secure_soc_startInit()**.
9. Funkcja **gsk_secure_soc_read()** odbiera komunikat od wątku procesu roboczego przy użyciu sesji chronionej.
10. Funkcja **gsk_secure_soc_close()** kończy sesję chronioną.
11. Funkcja **gsk_environment_close()** zamyka środowisko SSL.
12. Funkcja **close()** kończy połączenie.

```
/* Program serwera GSK używający ID aplikacji         */

/* Program zakłada, że identyfikator aplikacji został */
/* zarejestrowany, a certyfikat został przypisany     */
/* do ID aplikacji.                                   */
/*                                                    */
/* Brak parametrów, trochę komentarzy i wiele wartości*/
/* wpisanych w kodzie, aby przykład był prosty.       */

/* Użyj następującej komendy, aby utworzyć program    */
/* skonsolidowany:                                    */
/* 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;   /* uchwyt środowiska chronionego */
```

```
gsk_handle my_session_handle=NULL;    /* uchwyt sesji chronionej  */

struct sockaddr_in address;
int buf_len, rc = 0, sd = -1;
int amtWritten, amtRead;
char buff1[1024];
char buff2[1024];

/*  Zapisany w kodzie adres IP (należy zmienić na adres serwera) */
char addr[16] = "1.1.1.1";

/*********************************************/
/* Wszystkie komendy są uruchamiane w pętli  */
/* do/while, dzięki czemu czyszczenie odbywa */
/* się na końcu.                             */
/*********************************************/
do
{
  /* otwiera środowisko gsk */
  rc = errno = 0;
  rc = gsk_environment_open(&my_env_handle);
  if (rc != GSK_OK)
  {
    printf("Niepowodzenie gsk_environment_open() z kodem powrotu = %d kod błędu = %d.\n",
           rc,errno);
    printf("kod powrotu %d oznacza %s\n", rc, gsk_strerror(rc));
    break;
  }

  /* ustawia ID aplikacji */
  rc = errno = 0;
  rc = gsk_attribute_set_buffer(my_env_handle,
                                GSK_OS400_APPLICATION_ID,
                                "MY_CLIENT_APP",
                                13);
if (rc != GSK_OK)
  {
    printf("Niepowodzenie gsk_attribute_set_buffer() z kodem powrotu = %d kod błędu = %d.\n",
           rc,errno);
    printf("kod powrotu %d oznacza %s\n", rc, gsk_strerror(rc));
    break;
  }

  /* ustawia tę stronę jako klienta (domyślnie) */
  rc = errno = 0;
  rc = gsk_attribute_set_enum(my_env_handle,
                              GSK_SESSION_TYPE,
                              GSK_CLIENT_SESSION);
  if (rc != GSK_OK)
  {
    printf("Niepowodzenie gsk_attribute_set_enum() z kodem powrotu = %d kod błędu = %d.\n",
           rc,errno);
    printf("kod powrotu %d oznacza %s\n", rc, gsk_strerror(rc));
    break;
  }

  /* Protokoły SSL_V2, SSL_V3 i TLS_V1 są włączone domyślnie. */
  /* W tym przykładzie wyłączymy protokół SSL_V2.             */
  rc = errno = 0;
  rc = gsk_attribute_set_enum(my_env_handle,
                              GSK_PROTOCOL_SSLV2,
                              GSK_PROTOCOL_SSLV2_OFF);
  if (rc != GSK_OK)
  {
    printf("Niepowodzenie gsk_attribute_set_enum() z kodem powrotu = %d kod błędu = %d.\n",
           rc,errno);
    printf("kod powrotu %d oznacza %s\n", rc, gsk_strerror(rc));
```

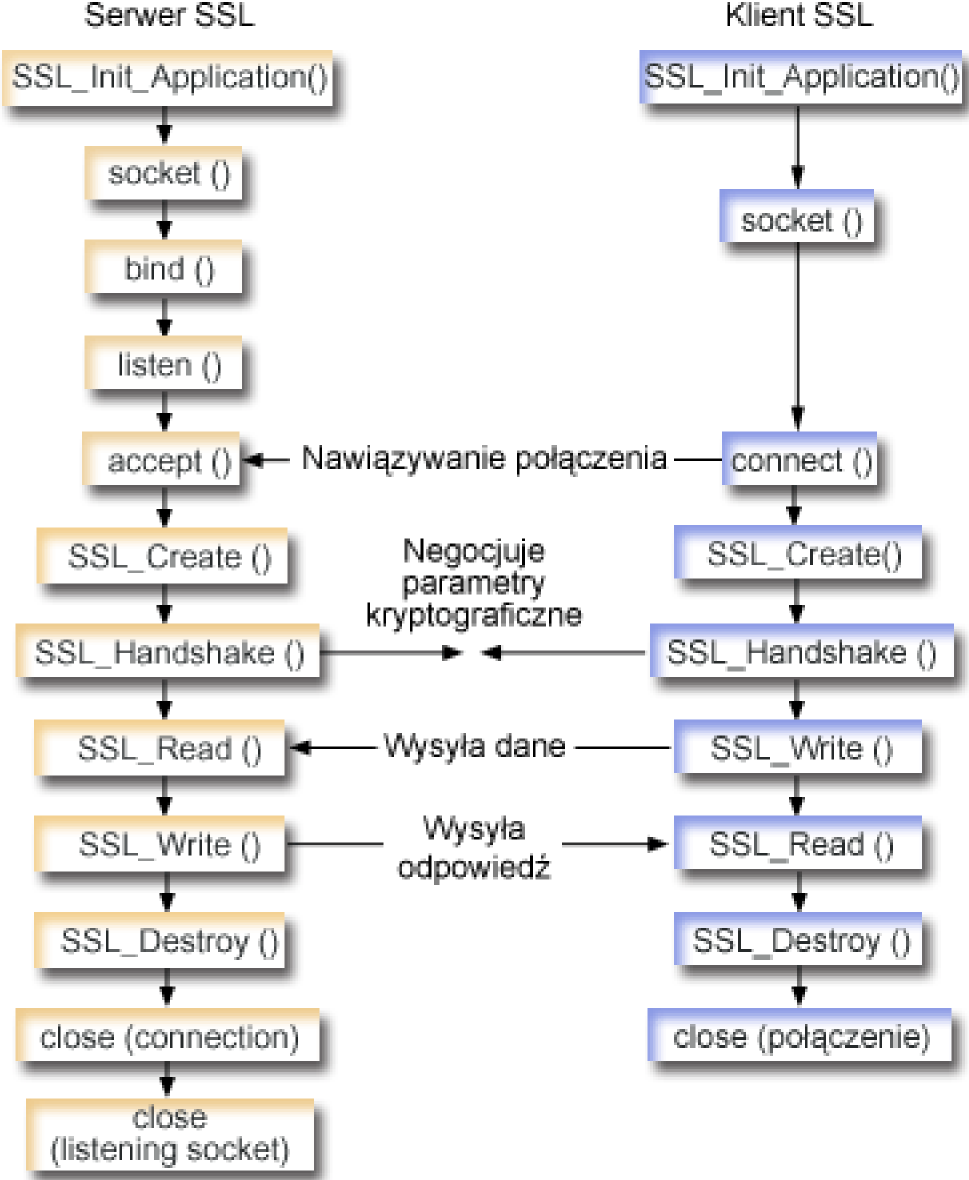
```
  break;
}

/* Określa, jakiego pakietu szyfrującego użyć. Domyślnie włączona jest */
/* domyślna lista szyfrowania. W tym przykładzie użyjemy jednego.      */
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("Niepowodzenie gsk_attribute_set_buffer() z kodem powrotu = %d kod błędu = %d.\n",
         rc,errno);
  printf("kod powrotu %d oznacza %s\n", rc, gsk_strerror(rc));
  break;
}

/* Zainicjowanie chronionego środowiska */
rc = errno = 0;
rc = gsk_environment_init(my_env_handle);
if (rc != GSK_OK)
{
  printf("Niepowodzenie gsk_environment_init() z kodem powrotu = %d kod błędu = %d.\n",
         rc,errno);
  printf("kod powrotu %d oznacza %s\n", rc, gsk_strerror(rc));
  break;
}

/* inicjowanie gniazda, które będzie użyte do nasłuchiwania */
sd = socket(AF_INET, SOCK_STREAM, 0);
if (sd < 0)
{
  perror("Niepowodzenie socket()");
  break;
}

/* połączenie z serwerem za pomocą ustawionego numeru portu */
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("Niepowodzenie connect()");
  break;
}

/* otwarcie sesji chronionej */
rc = errno = 0;
rc = gsk_secure_soc_open(my_env_handle, &my_session_handle);
if (rc != GSK_OK)
{
  printf("Niepowodzenie gsk_secure_soc_open() z kodem powrotu = %d kod błędu = %d.\n",
         rc,errno);
  printf("kod powrotu %d oznacza %s\n", rc, gsk_strerror(rc));
  break;
}

/* powiązanie gniazda z sesją chronioną */
rc=errno=0;
rc = gsk_attribute_set_numeric_value(my_session_handle,
                                     GSK_FD,
                                     sd);
if (rc != GSK_OK)
{
```

```
    printf("Niepowodzenie gsk_attribute_set_numeric_value() z kodem powrotu = %d ", rc);
    printf("i numerem błędu = %d.\n", errno);
    printf("kod powrotu %d oznacza %s\n", rc, gsk_strerror(rc));
    break;
  }

  /* inicjowanie uzgodnienia SSL */
  rc = errno = 0;
  rc = gsk_secure_soc_init(my_session_handle);
  if (rc != GSK_OK)
  {
    printf("Niepowodzenie gsk_secure_soc_open() z kodem powrotu = %d kod błędu = %d.\n",
           rc,errno);
    printf("kod powrotu %d oznacza %s\n", rc, gsk_strerror(rc));
    break;
  }

  /* Funkcja memset zeruje bufor szesnastkowo      */
  memset((char *) buff1, 0, sizeof(buff1));

  /* wysłanie komunikatu do serwera w ramach bezpiecznej sesji */
  strcpy(buff1,"Test of gsk_secure_soc_write \n\n");

  /* wysłanie komunikatu do klienta w ramach bezpiecznej sesji */
  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("Kod powrotu gsk_secure_soc_write() = %d, numer błędu = %d.\n",rc,errno);
      printf("kod powrotu %d oznacza %s\n", rc, gsk_strerror(rc));
      break;
    }
    else
    {
      printf("Funkcja gsk_secure_soc_write() nie zapisała wszystkich danych.\n");
      break;
    }
  }

  /* wyświetlenie wyników na ekranie */
  printf("Funkcja gsk_secure_soc_write() zapisała bajtów: %d ...\n", amtWritten);
  printf("%s\n",buff1);

/* Funkcja memset zeruje bufor szesnastkowo      */
  memset((char *) buff2, 0x00, sizeof(buff2));

  /* odebranie komunikatu od klienta w ramach bezpiecznej sesji */
  amtRead = 0;
  rc = gsk_secure_soc_read(my_session_handle, buff2, sizeof(buff2), &amtRead);

  if (rc != GSK_OK)
  {
    printf("Kod powrotu gsk_secure_soc_read() = %d, numer błędu = %d.\n",rc,errno);
    printf("kod powrotu %d oznacza %s\n", rc, gsk_strerror(rc));
    break;
  }

  /* wyświetlenie wyników na ekranie */
  printf("Funkcja gsk_secure_soc_read() odebrała bajtów: %d, oto one:\n",
         amtRead);
  printf("%s\n",buff2);


} while(FALSE);
```

```
  /* wyłączenie obsługi SSL dla gniazda */
  if (my_session_handle != NULL)
    gsk_secure_soc_close(&my_session_handle);

  /* wyłączenie środowiska SSL */
  if (my_env_handle != NULL)
    gsk_environment_close(&my_env_handle);

  /* zamknięcie połączenia */
  if (sd > -1)
    close(sd);

  return;
}
```

Przykład: chroniony serwer używający funkcji API SSL_

Tworząc aplikacje chronione można oprócz funkcji API GSKit używać funkcji API SSL_. Są one rodzime w systemie operacyjnym iSeries. Podobnie jak w przypadku funkcji API GSKit, do bezpiecznej wymiany danych serwer musi udostępnić poprawny certyfikat. Poniższa ilustracja przedstawia funkcje API gniazd i SSL_ używane do tworzenia serwera chronionego. Jeśli aplikacje chronione mają być używane na różnych platformach IBM eserver, należy użyć funkcji API GSKit.



Przebieg zdarzeń w gnieździe: chroniony serwer używający funkcji API SSL_

Poniżej opisano relacje pomiędzy funkcjami API umożliwiającymi pracę serwera SSL i jego komunikację z klientem SSL:

1. Wywołanie funkcji **SSL_Init()** lub **SSL_Init_Application()** w celu zainicjowania środowiska pracy dla przetwarzania SSL i w celu określenia informacji o ochronie SSL dla wszystkich sesji SSL, które będą uruchamiane w bieżącym zadaniu. Należy użyć tylko jednej z tych funkcji API. Zalecane jest użycie funkcji API **SSL_Init_Application()**.

   Uwaga: Poniższy program przykładowy używa funkcji API **SSL_Init_Application**.
2. Serwer wywołuje funkcję **socket()** w celu uzyskania deskryptora gniazda.
3. Serwer wywołuje funkcje **bind()**, **listen()** i **accept()**, aby uaktywnić połączenie dla programu serwera.
4. Serwer wywołuje funkcję **SSL_Create()**, aby włączyć obsługę SSL dla podłączonego gniazda.

5. Serwer wywołuje funkcję **SSL_Handshake()**, aby zainicjować uzgadnianie SSL parametrów szyfrujących.
6. Serwer wywołuje funkcje **SSL_Write()** i **SSL_Read()**, aby wysłać i odebrać dane.
7. Serwer wywołuje funkcję **SSL_Destroy()** w celu wyłączenia obsługi SSL dla gniazda.
8. Serwer wywołuje funkcję **close()** w celu usunięcia podłączonych gniazd.

Przebieg zdarzeń w gnieździe: chroniony klient używający funkcji API SSL_

1. Wywołanie funkcji **SSL_Init()** lub **SSL_Init_Application()** w celu zainicjowania środowiska pracy dla przetwarzania SSL i w celu określenia informacji o ochronie SSL dla wszystkich sesji SSL, które będą uruchamiane w bieżącym zadaniu. Należy użyć tylko jednej z tych funkcji API. Zalecane jest użycie funkcji API **SSL_Init_Application**.

   Uwaga: Poniższy program przykładowy używa funkcji API **SSL_Init_Application**.
2. Klient wywołuje funkcję **socket()** w celu uzyskania deskryptora gniazda.
3. Klient wywołuje funkcję **connect()** w celu uaktywnienia połączenia dla programu klienta.
4. Klient wywołuje funkcję **SSL_Create()**, aby włączyć obsługę SSL dla podłączonego gniazda.
5. Klient wywołuje funkcję **SSL_Handshake()** , aby zainicjować uzgadnianie SSL parametrów szyfrujących.
6. Klient wywołuje funkcje **SSL_Read()** i **SSL_Write()** w celu odebrania i wysłania danych.
7. Klient wywołuje funkcję **SSL_Destroy()** w celu wyłączenia obsługi SSL dla gniazda.
8. Klient wywołuje funkcję **close()** w celu usunięcia podłączonych gniazd.

Uwaga: W przykładzie użyto rodziny adresów AF_INET; można go jednak zmodyfikować, aby została użyta rodzina adresów AF_INET6.

Informacje dotyczące wykorzystania przykładowych kodów zawiera sekcja Informacje dotyczące kodu.

```
/* Program serwera SSL używający funkcji SSL_Init_Application */

/* Przyjmuje się, że ID aplikacji jest już           */
/* zarejestrowane i powiązane z certyfikatem.        */
/*                                                   */
/* Brak parametrów, trochę komentarzy i wiele wartości*/
/* wpisanych w kodzie, aby przykład był prosty.      */

/* Użyj następującej komendy, aby utworzyć program    */
/* skonsolidowany:                                   */
/* 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];

    /*  tylko jeden zestaw algorytmów szyfrowania      */
    unsigned short int cipher = SSL_RSA_WITH_RC4_128_SHA;

    void * malloc_ptr = (void *) NULL;
```

```
unsigned int malloc_size = 8192;

/* memset - szesnastkowe zerowanie struktury sslinit */
memset((char *)&sslinit, 0, sizeof(sslinit));

/* wypełnianie wartościami struktury sslinit */
sslinit.applicationID = "MY_SERVER_APP";
sslinit.applicationIDLen = 13;
sslinit.localCertificate = NULL;
sslinit.localCertificateLen = 0;
sslinit.cipherSuiteList = NULL;
sslinit.cipherSuiteListLen = 0;

/* przydzielanie i ustawianie wskaźników dla buforu certyfikatu */
malloc_ptr = (void*) malloc(malloc_size);
sslinit.localCertificate = (unsigned char*) malloc_ptr;
sslinit.localCertificateLen = malloc size;

/* inicjowanie wywołania SSL SSL_Init_Application */
rc = SSL_Init_Application(&sslinit);
if (rc != 0)
{
  printf("Niepowodzenie SSL_Init_Application() z kodem powrotu = %d i kodem błędu = %d.\n",
         rc,errno);
  return;
}

/* inicjowanie gniazda, które będzie użyte do nasłuchiwania */
lsd = socket(AF_INET, SOCK_STREAM, 0);
if (lsd < 0)
{
  perror("Niepowodzenie socket()");
  return;
}

/* ustawienie gniazda do natychmiastowego ponownego użycia */
rc = setsockopt(lsd, SOL_SOCKET,
                SO_REUSEADDR,
                (char *)&on,
                sizeof(on));
if (rc < 0)
{
  perror("Niepowodzenie setsockopt()");
  return;
}

/* powiązanie do adresu lokalnego serwera */
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("Niepowodzenie bind()");
  close(lsd);
  return;
}

/* uaktywnienie gniazda dla przychodzących połączeń klienta */
listen(lsd, 5);
if (rc < 0)
{
  perror("Niepowodzenie listen()");
  close(lsd);
  return;
}
```

```
/* akceptacja przychodzącego połączenia klienta */
al = sizeof(address);
sd = accept(lsd, (struct sockaddr *) &address, &al);
if (sd < 0)
{
  perror("Niepowodzenie accept()");
  close(lsd);
  return;
}

/* uaktywnienie obsługi SSL dla gniazda */
sslh = SSL_Create(sd, SSL_ENCRYPT);
if (sslh == NULL)
{
  printf("Niepowodzenie SSL_Create() z errno = %d.\n", errno);
  close(lsd);
  close(sd);
  return;
}

/* ustawienie parametrów dla uzgadniania */
sslh -> protocol = 0;
sslh -> timeout = 0;
sslh -> cipherSuiteList = &cipher;
sslh -> cipherSuiteListLen = 1;

/* inicjowanie uzgodnienia SSL */
rc = SSL_Handshake(sslh, SSL_HANDSHAKE_AS_SERVER);
if (rc != 0)
{
  printf("Niepowodzenie SSL_Handshake() z kodem powrotu = %d i kodem błędu = %d.\n",
      rc,errno);
  SSL_Destroy(sslh);
  close(lsd);
  close(sd);
  return;
}

/* Funkcja memset zeruje bufor szesnastkowo      */
memset((char *) buff, 0, sizeof(buff));

/* odebranie komunikatu od klienta w ramach bezpiecznej sesji */
rc = SSL_Read(sslh, buff, sizeof(buff));
if (rc < 0)
{
  printf("SSL_Read() rc = %d i errno = %d.\n",rc,errno);
  rc = SSL_Destroy(sslh);
  if (rc != 0)
    printf("SSL_Destroy() rc = %d i errno = %d.\n",rc,errno);
  close(lsd);
  close(sd);
  return;
}

/* wyświetlenie wyników na ekranie */
printf("SSL_Read() odczytała ...\n");
printf("%s\n",buff);

/* wysłanie komunikatu do klienta w ramach bezpiecznej sesji */
buf_len = strlen(buff);
rc = SSL_Write(sslh, buff, buf_len);
if (rc != buf_len)
{
  if (rc < 0)
  {
    printf("Niepowodzenie SSL_Write() z rc = %d.\n",rc);
```

```
        SSL_Destroy(sslh);
        close(lsd);
        close(sd);
        return;
      }
      else
      {
        printf("SSL_Write() nie zapisała wszystkich danych.\n");
        SSL_Destroy(sslh);
        close(lsd);
        close(sd);
        return;
      }
    }

    /* wyświetlenie wyników na ekranie */
    printf("SSL_Write() zapisała ...\n");
    printf("%s\n",buff);

    /* wyłączenie obsługi SSL dla gniazda */
    SSL_Destroy(sslh);

    /* zamknięcie połączenia */
    close(sd);

    /* zamknięcie gniazda nasłuchującego */
    close(lsd);

    return;
}
```

Przykład: chroniony klient używający funkcji API SSL_

Oprócz funkcji API GSKit, gniazda OS/400 obsługują tradycyjne funkcje API SSL_. Funkcje te nawiązują chronione połączenie pomiędzy rodzimym serwerem iSeries a aplikacjami klienckimi. Ilustracja opisująca przebieg zdarzeń w gnieździe w tym programie i aplikacji serwera, z którą się komunikuje, znajduje się w sekcji Przykład: chroniony serwer używający funkcji API SSL_. Poniższy przykład to aplikacja kliencka używająca funkcji API SSL_ do komunikowania się z aplikacją serwera używającą funkcji API SSL_:

Informacje dotyczące wykorzystania przykładowych kodów zawiera sekcja Informacje dotyczące kodu.

```
/* Program klienta SSL używający funkcji SSL_Init_Application */

/* Przyjmuje się, że ID aplikacji jest już           */
/* zarejestrowane i powiązane z certyfikatem.        */
/*                                                   */
/* Brak parametrów, trochę komentarzy i wiele wartości*/
/* wpisanych w kodzie, aby przykład był prosty.      */

/* Użyj następującej komendy, aby utworzyć program    */
/* skonsolidowany:                                   */
/* 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>

/*  dla uproszczenia bez parametrów  */
```

```
void main(void)
{
    SSLHandle *sslh;
    SSLInitApp sslinit;
    struct sockaddr_in address;
    int buf_len, rc = 0, sd;
    char buff1[1024];
    char buff2[1024];

    /*  tylko jeden zestaw algorytmów szyfrowania      */
    unsigned short int cipher = SSL_RSA_WITH_RC4_128_SHA;

    /*  wpisany na stałe w kodzie adres IP             */
    char addr[12] = "16.35.146.84";

    void * malloc_ptr = (void *) NULL;
    unsigned int malloc_size = 8192;

    /* memset - szesnastkowe zerowanie struktury sslinit */
    memset((char *)&sslinit, 0, sizeof(sslinit));

    /* wypełnianie wartościami struktury sslinit */
    /* z użyciem istniejącego ID aplikacji       */
    sslinit.applicationID = "MY_CLIENT_APP";
    sslinit.applicationIDLen = 13;
    sslinit.localCertificate = NULL;
    sslinit.localCertificateLen = 0;
    sslinit.cipherSuiteList = NULL;
    sslinit.cipherSuiteListLen = 0;

    /* przydzielanie i ustawianie wskaźników dla buforu certyfikatu */
    malloc_ptr = (void*) malloc(malloc_size);
    sslinit.localCertificate = (unsigned char*) malloc_ptr;
    sslinit.localCertificateLen = malloc size;

    /* inicjowanie wywołania SSL SSL_Init_Application */
    rc = SSL_Init_Application(&sslinit);
    if (rc != 0)
    {
      printf("Niepowodzenie SSL_Init_Application() z kodem powrotu = %d i kodem błędu = %d.\n",
           rc,errno);
      return;
    }

    /* inicjowanie gniazda */
    sd = socket(AF_INET, SOCK_STREAM, 0);
    if (sd < 0)
    {
      perror("Niepowodzenie socket()");
      return;
    }

    /* uaktywnienie obsługi SSL dla gniazda */
    sslh = SSL_Create(sd, SSL_ENCRYPT);
    if (sslh == NULL)
    {
      printf("Niepowodzenie SSL_Create() z errno = %d.\n", errno);
      close(sd);
      return;
    }

    /* połączenie z serwerem za pomocą ustawionego numeru portu */
    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("Niepowodzenie connect()");
  close(sd);
  return;
}

/* przygotowanie do wywołania uzgodnienia, ustawianie algorytmu */
sslh -> protocol = 0;
sslh -> timeout = 0;
sslh -> cipherSuiteList = &cipher;
sslh -> cipherSuiteListLen = 1;

/* inicjowanie uzgodnienia SSL - jako KLIENT */
rc = SSL_Handshake(sslh, SSL_HANDSHAKE_AS_CLIENT);
if (rc != 0)
{
  printf("Niepowodzenie SSL_Handshake() z kodem powrotu = %d i kodem błędu = %d.\n",
       rc,errno);
  close(sd);
  return;
}

/* wysłanie komunikatu do serwera w ramach bezpiecznej sesji */
strcpy(buff1,"Test funkcji SSL_Write \n\n");
buf_len = strlen(buff1);
rc = SSL_Write(sslh, buff1, buf_len);
if (rc != buf_len)
{
  if (rc < 0)
  {
   printf("Niepowodzenie SSL_Write() z rc = %d i errno = %d.\n",rc, errno);
    SSL_Destroy(sslh);
    close(sd);
    return;
  }
  else
  {
    printf("SSL_Write() nie zapisała wszystkich danych.\n");
    SSL_Destroy(sslh);
    close(sd);
    return;
  }
}

/* wyświetlenie wyników na ekranie */
printf("SSL_Write() zapisała ...\n");
printf("%s\n",buff1);

memset((char *) buff2, 0x00, sizeof(buff2));

/* odebranie komunikatu z serwera w ramach bezpiecznej sesji */
rc = SSL_Read(sslh, buff2, buf_len);
if (rc < 0)
{
  printf("Niepowodzenie SSL_Read() z rc = %d.\n",rc);
  SSL_Destroy(sslh);
  close(sd);
  return;
}

/* wyświetlenie wyników na ekranie */
printf("SSL_Read() odczytała ...\n");
printf("%s\n",buff2);

/* wyłączenie obsługi SSL dla gniazda */
SSL_Destroy(sslh);
```

```
    /* zamknięcie połączenia przez zamknięcie lokalnego gniazda */
    close(sd);
    return;
}
```

Przykład: procedury sieciowe obsługujące ochronę wątków i używające funkcji gethostbyaddr_r()

Poniżej przedstawiono przykład programu, w którym wykorzystano funkcję **gethostbyaddr_r()**. Wszystkie pozostałe procedury, które używają nazw zakończonych na ″_r″, mają podobną semantykę i także obsługują chronione wątki. Ten przykładowy program pobiera adres IP w notacji dziesiętnej z kropkami i drukuje nazwę hosta.

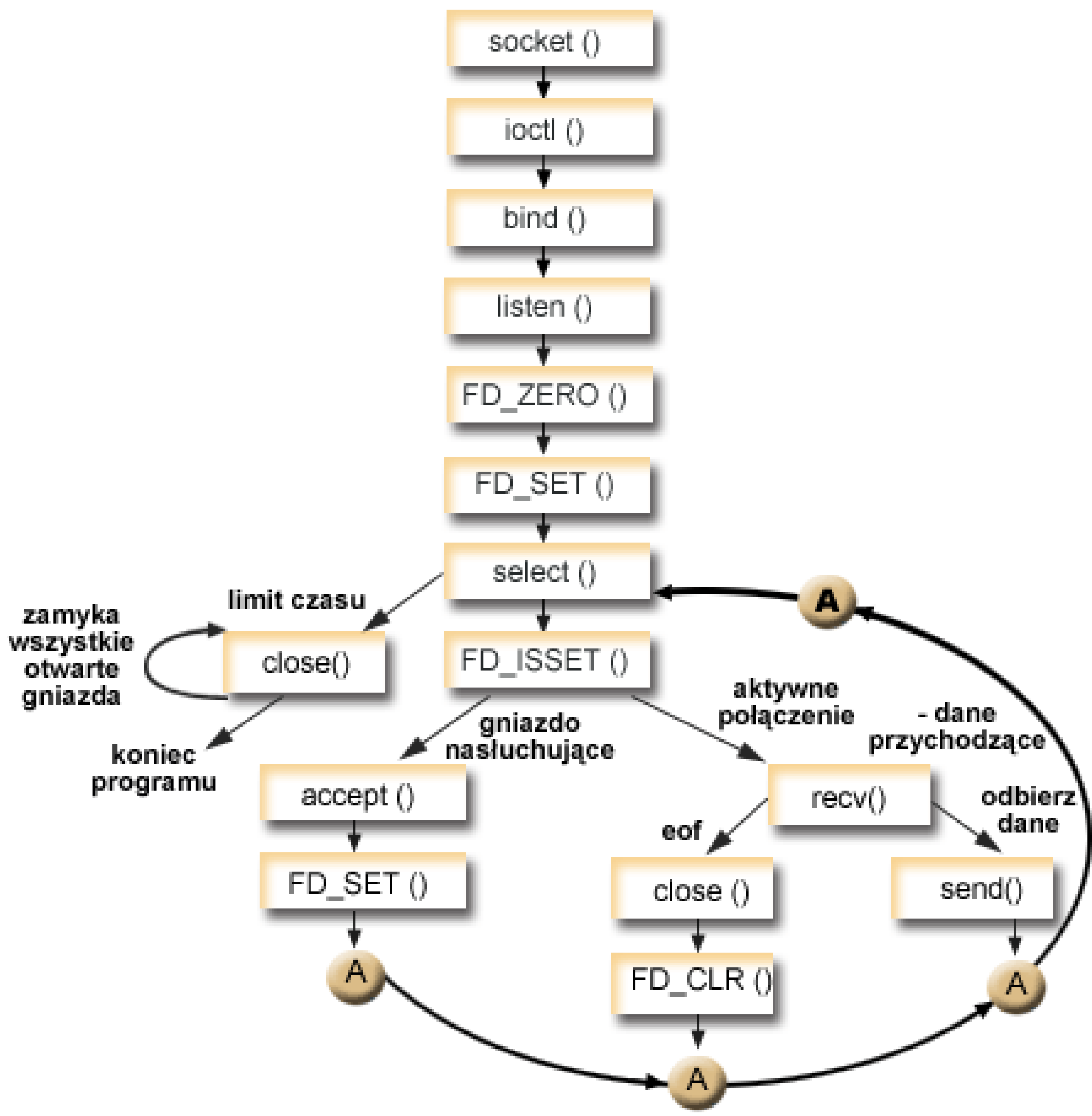
Informacje dotyczące wykorzystania przykładowych kodów zawiera sekcja Informacje dotyczące kodu.

```
/********************************************************/
/* Pliki nagłówkowe                                     */
/********************************************************/
#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
/********************************************************/
/* Przekaż parametr, który jest adresem IP w notacji    */
/* dziesiętnej z kropkami. Nazwa hosta zostanie         */
/* wyświetlona, jeśli będzie znaleziona; w przeciwnym   */
/* razie wyświetl komunikat 'hosta nie znaleziono'.     */
/********************************************************/
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];

    /********************************************************/
    /* Sprawdź liczbę przekazanych  argumentów              */
    /********************************************************/
    if (argc != NUMPARMS)
     {
      printf("Zła liczba wpisanych argumentów parametrów\n");
      exit(-1);
     }
    /********************************************************/
    /* Uzyskaj adresowalność przekazanych parametrów        */
    /********************************************************/
    strcpy(dotted_decimal_address, argv[1]);

    /********************************************************/
    /* Zainicjuj pole struktury                             */
    /* hostent_data.host_control_blk szesnastkowymi zerami  */
    /* zanim zostanie użyte. Jeśli wymagasz zgodności z innymi*/
    /* platformami, musisz zainicjować całą strukturę       */
    /* hostent_data szesnastkowymi zerami.                  */
    /********************************************************/
    /* Zainicjuj strukturę hostent_data do szesnastkowych 00  */
    /********************************************************/
    memset(&hst_ent_data,HEX00,sizeof(struct hostent_data));
```

```
    /*******************************************************/
    /* Przetłumacz adres internetowy z postaci dziesiętnej    */
    /* z kropkami do formatu 32-bitowego adresu IP.           */
    /*******************************************************/
    internet_address.s_addr=inet_addr(dotted_decimal_address);

    /*******************************************************/
    /* Uzyskaj nazwę hosta                                     */
    /*******************************************************/
    /*******************************************************/
    /* UWAGA: gethostbyaddr_r() zwraca liczbę całkowitą.      */
    /* Oto możliwe wartości:                                   */
    /*   -1 (wywołanie nie powiodło się)                       */
    /*   0 (wywołanie powiodło się)                            */
    /*******************************************************/
    rc=gethostbyaddr_r((char *) &internet_address,
                       sizeof(struct in_addr), AF_INET,
                       &hst_ent, &hst_ent_data);

    if (rc== -1)
{
         printf("Nie znaleziono nazwy hosta\n");
         exit(-1);
       }
    else
       {
        /**************************************************/
        /*   Skopiuj nazwę hosta do buforu danych wyjściowych*/
        /**************************************************/
          (void) memcpy((void *) host_name,
        /**************************************************/
        /* Wszystkie rezultaty należy adresować poprzez     */
        /* strukturę hostent (hst_ent).                     */
        /* UWAGA: Struktura danych hostent_data             */
        /* (hst_ent_data) jest tylko repozytorium           */
        /* danych używanym do obsługi struktury             */
        /* hostent. Aplikacje powinny traktować strukturę   */
        /* hostent_data jako obszar przechowywania danych   */
        /* poziomu hosta, do których                        */
        /* nie muszą mieć dostępu.                          */
        /**************************************************/
                        (void *) hst_ent.h_name,
                        MAXHOSTNAMELEN);
        /**************************************************/
        /* Wydrukuj nazwę hosta                             */
        /**************************************************/
          printf("Nazwa hosta to %s\n", host_name);

         }
       exit(0);
     }
```

Przykład: nieblokujące operacje we/wy i funkcja select()

Poniższy program przykładowy wykorzystuje nieblokujące operacje we/wy i funkcję API select(). W sekcji Przykład: ogólny program klienta znajduje się przykład kodu dla typowego zadania klienta, którego można użyć z niniejszym przykładem.



Przebieg zdarzeń w gnieździe: serwer używający nieblokujących operacji we/wy i funkcji select()
W przykładzie użyto następujących wywołań funkcji:

1. Funkcja **socket()** zwraca deskryptor gniazda odpowiadający punktowi końcowemu. Instrukcja ta informuje również, że dla tego gniazda zostanie użyta rodzina adresów INET (Internet Protocol) z transportem TCP transport (SOCK_STREAM).
2. Funkcja **ioctl()** umożliwia ponowne użycie adresu lokalnego po restarcie serwera, zanim upłynie wymagany czas oczekiwania. W tym przykładzie ustawia gniazdo na nieblokujące. Wszystkie gniazda dla połączeń przychodzących będą również nieblokujące, ponieważ będą dziedziczyć ten stan od gniazda nasłuchującego.
3. Po utworzeniu deskryptora gniazda funkcja **bind()** pobiera unikalną nazwę gniazda.
4. Funkcja **listen()** umożliwia serwerowi przyjęcie połączenia przychodzącego od klienta.
5. Serwer używa funkcji **accept()** do zaakceptowania żądania połączenia przychodzącego. Wywołanie funkcji **accept()** zostanie zablokowane na nieokreślony czas oczekiwania na połączenie przychodzące.

6. Funkcja **select()** powoduje, że proces oczekuje na wystąpienie zdarzenia, po którym kontynuuje działanie. W tym przykładzie funkcja **select ()** zwraca liczbę odpowiadającą deskryptorom gniazda, które są gotowe do przetwarzania.

   0 Wskazuje, że nastąpi przekroczenie limitu czasu procesu. W tym przykładzie limit czasu ustawiono na 30 sekund.

   −1 Wskazuje, że proces nie powiódł się.

   1 Wskazuje, że tylko jeden deskryptor jest gotowy do przetwarzania. W tym przykładzie zwrócenie wartości 1 powoduje, że FD_ISSET i kolejne wywołania gniazd zostaną zakończone tylko raz.

   n Wskazuje, że na przetwarzanie czeka wiele deskryptorów. W tym przykładzie zwrócenie wartości n powoduje, że FD_ISSET i dalszy kod zapętlają się i kończą obsługę żądań w kolejności ich odebrania przez serwer.
7. Funkcje **accept()** i **recv()** kończą działanie po zwróceniu wartości EWOULDBLOCK.
8. Funkcja **send()** odsyła dane do klienta.
9. Funkcja **close()** zamyka wszystkie otwarte deskryptory gniazd.

Informacje dotyczące wykorzystania przykładowych kodów zawiera sekcja Informacje dotyczące kodu.

```
#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;

   /*************************************************************/
   /* Tworzy gniazdo strumienia AF_INET do odbierania połączeń  */
   /* przychodzących.                                           */
   /*************************************************************/
   listen_sd = socket(AF_INET, SOCK_STREAM, 0);
   if (listen_sd < 0)
   {
      perror("Niepowodzenie socket()");
      exit(-1);
   }

   /*************************************************************/
   /* Umożliwia ponowne użycie deskryptora gniazda              */
   /*************************************************************/
   rc = setsockopt(listen_sd, SOL_SOCKET,  SO_REUSEADDR,
                   (char *)&on, sizeof(on));
   if (rc < 0)
   {
      perror("Niepowodzenie setsockopt()");
      close(listen_sd);
```

```
      exit(-1);
   }

   /*************************************************************/
   /* Ustawienie gniazda na nieblokujące. Wszystkie gniazda dla */
   /* połączeń przychodzących będą również nieblokujące, ponie- */
   /* waż będą dziedziczyć ten stan od gniazda nasłuchującego.  */
   /*************************************************************/
   rc = ioctl(listen_sd, FIONBIO, (char *)&on);
   if (rc < 0)
   {
      perror("Niepowodzenie ioctl()");
      close(listen_sd);
      exit(-1);
   }

   /*************************************************************/
   /* Powiąż gniazdo                                            */
   /*************************************************************/
   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("Niepowodzenie bind()");
      close(listen_sd);
      exit(-1);
   }

   /*************************************************************/
   /* Ustawia nasłuchiwanie parametru backlog.                  */
   /*************************************************************/
   rc = listen(listen_sd, 32);
   if (rc < 0)
   {
      perror("Niepowodzenie listen()");
      close(listen_sd);
      exit(-1);
   }

   /*************************************************************/
   /* Inicjuje główny fd_set                                    */
   /*************************************************************/
   FD_ZERO(&master_set);
   max_sd = listen_sd;
   FD_SET(listen_sd, &master_set);

   /*************************************************************/
   /* Inicjuje strukturę timeval wartością 3 min. Brak          */
   /* aktywności w tym czasie spowoduje zakończenie programu.   */
   /*************************************************************/
   timeout.tv_sec  = 3 * 60;
   timeout.tv_usec = 0;

   /*************************************************************/
   /* Pętla oczekiwania na przychodzące połączenia lub dane     */
   /* dla dowolnego połączonego gniazda.                        */
   /*************************************************************/
   do
   {
      /**********************************************************/
      /* Kopiuje główny fd_set do roboczego fd_set.             */
      /**********************************************************/
      memcpy(&working_set, &master_set, sizeof(master_set));
```

```
/**********************************************************/
/* Wywołuje select() i czeka 5 min na zakończenie.        */
/**********************************************************/
printf("Oczekiwanie na select()...\n");
rc = select(max_sd + 1, &working_set, NULL, NULL, &timeout);

/**********************************************************/
/* Sprawdzenie, czy wywołanie funkcji select się powiodło.*/
/**********************************************************/
if (rc < 0)
{
   perror("Niepowodzenie select()");
   break;
}

/**********************************************************/
/* Sprawdzenie, czy nie upłynął czas oczekiwania 5 min.   */
/**********************************************************/
if (rc == 0)
{
   printf("Przekroczenie czasu dla select(). Koniec programu.\n");
   break;
}

/**********************************************************/
/* Przynajmniej jeden deskryptor jest czytelny. Trzeba    */
/* sprawdzić, który.                                      */
/**********************************************************/
desc_ready = rc;
for (i=0; i <= max_sd  &&  desc_ready > 0; ++i)
{
   /*******************************************************/
   /* Sprawdzenie, czy deskryptor jest gotowy.            */
   /*******************************************************/
   if (FD_ISSET(i, &working_set))
   {
      /****************************************************/
      /* Znaleziono czytelny deskryptor - o jeden mniej   */
      /* do znalezienia. Czynność tę powtarza się, więc   */
      /* po znalezieniu wszystkich gotowych deskryptorów  */
      /* można przestać przeszukiwać zestaw roboczy.      */
      /*                                                  */
      /****************************************************/
      desc_ready -= 1;

      /****************************************************/
      /* Sprawdzenie, czy to nasłuchujące gniazdo.        */
      /****************************************************/
      if (i == listen_sd)
      {
         printf("Gniazdo nasłuchujące jest gotowe\n");
         /*************************************************/
         /* Akceptowanie wszystkich połączeń przychodz.,  */
         /* które znajdują się w kolejce gniazda          */
         /* nasłuchującego przed ponownym wywołaniem      */
         /* funkcji select.                               */
         /*************************************************/
         do
         {
            /**********************************************/
            /* Akceptowanie każdego połączenia przychodz. */
            /* Jeśli akceptowanie nie powiedzie się z     */
            /* wartością EWOULDBLOCK, to znaczy, że       */
            /* zostały zaakceptowane wszystkie. Każde inne*/
            /* niepowodzenie akceptowania wymusza         */
            /* zatrzymanie serwera.                       */
```

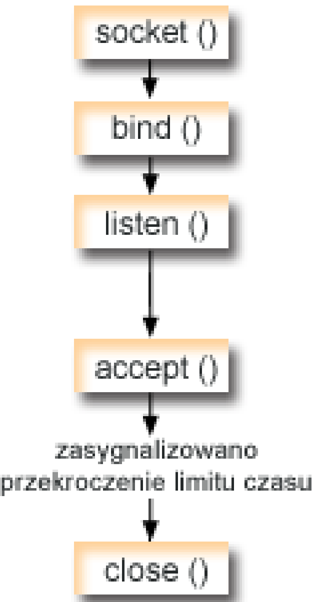
```
         /**********************************************/
         new_sd = accept(listen_sd, NULL, NULL);
         if (new_sd < 0)
         {
            if (errno != EWOULDBLOCK)
            {
               perror("Niepowodzenie accept()");
               end_server = TRUE;
            }
            break;
         }

         /**********************************************/
         /* Dodanie nowego połączenia przychodzącego   */
         /* do głównego zestawu operacji odczytu.      */
         /**********************************************/
         printf("Nowe połączenie przychodzące - %d\n", new_sd);
         FD_SET(new_sd, &master_set);
         if (new_sd > max_sd)
            max_sd = new_sd;

         /**********************************************/
         /* Powrót w pętli i akceptowanie nowego       */
         /* połączenia przychodzącego.                 */
         /**********************************************/
      } while (new_sd != -1);
   }

   /****************************************************/
   /* To nie jest gniazdo nasłuchujące, dlatego        */
   /* istniejące połączenie musi być czytelne          */
   /****************************************************/
   else
   {
      printf("Deskryptor %d jest czytelny\n", i);
      close_conn = FALSE;
      /*************************************************/
      /* Odbiór wszystkich danych przychodzących do    */
      /* tego gniazda przed powrotem w pętli i ponownym*/
      /* wywołaniem funkcji select.                    */
      /*************************************************/
      do
      {
         /**********************************************/
         /* Odbieranie danych dla tego połączenia,     */
         /* dopóki recv nie powiedzie się z wartością  */
         /* EWOULDBLOCK. Jeśli wystąpi inne            */
         /* niepowodzenie, połączenie zost. zamknięte. */
         /**********************************************/
         rc = recv(i, buffer, sizeof(buffer), 0);
         if (rc < 0)
         {
            if (errno != EWOULDBLOCK)
            {
               perror("Niepowodzenie recv()");
               close_conn = TRUE;
            }
            break;
         }

         /**********************************************/
         /* Sprawdzenie, czy połączenie nie zostało    */
         /* zamknięte przez klienta.                   */
         /**********************************************/
         if (rc == 0)
         {
            printf("  Połączenie zamknięte\n");
```

```
                  close_conn = TRUE;
                  break;
               }

               /**********************************************/
               /* Dane zostały odebrane.                     */
               /**********************************************/
               len = rc;
               printf("Otrzymano bajtów: %d\n", len);

               /**********************************************/
               /* Odesłanie danych do klienta                */
               /**********************************************/
               rc = send(i, buffer, len, 0);
               if (rc < 0)
               {
                  perror("Niepowodzenie send()");
                  close_conn = TRUE;
                  break;
               }

            } while (TRUE);

            /*************************************************/
            /* Jeśli opcja close_conn została włączona,      */
            /* trzeba wyczyścić aktywne połączenie. Procedura*/
            /* czyszcząca obejmuje usunięcie deskryptora z   */
            /* zestawu głównego i określenie nowej wartości  */
            /* maksymalnej deskryptora na podstawie bitów,   */
            /* które wciąż są włączone w zestawie głównym.   */
            /*                                               */
            /*************************************************/
            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;
               }
            }
         } /* Koniec "istniejące połączenie musi być czytelne"*/
      } /* Koniec "if (FD_ISSET(i, &working_set))" */
   } /* Koniec pętli poprzez wybierane deskryptory  */

} while (end_server == FALSE);

/*************************************************************/
/* Czyszczenie wszystkich otwartych gniazd                   */
/*************************************************************/
for (i=0; i <= max_sd; ++i)
{
   if (FD_ISSET(i, &master_set))
      close(i);
}
}
```

Przykład: używanie sygnałów z blokującymi funkcjami API gniazd

Sygnały powiadamiają o zablokowaniu procesu lub aplikacji. Udostępniają one limit czasu blokowania procesów. W tym przykładzie sygnał pojawia się po pięciu sekundach po wywołaniu funkcji **accept()**. Normalnie wywołanie to blokowałoby przez czas nieograniczony, ale alarm powoduje ograniczenie tego czasu do 5 sekund. Ponieważ zablokowane programy mogą pogarszać wydajność aplikacji lub serwera, można użyć sygnałów do zmniejszenia tego wpływu. Poniższy przykład pokazuje, w jaki sposób używać sygnałów z blokującymi funkcjami API gniazd.

Uwaga: Zamiast konwencjonalnego modelu preferowane jest użycie asynchronicznych operacji we/wy w modelu serwera z obsługą wątków. Więcej informacji o korzyściach płynących z zastosowania asynchronicznych operacji we/wy zawiera sekcja Asynchroniczne operacje we/wy. Przykład programu używającego asynchronicznych operacji we/wy znajduje się w sekcji Przykład: korzystanie z asynchronicznych operacji we/wy.



Przebieg zdarzeń w gnieździe: używanie sygnałów z blokowaniem gniazd
Opisane poniżej wywołania funkcji ilustrują sposób użycia sygnałów do powiadomienia aplikacji o tym, że gniazdo jest nieaktywne.

1. Funkcja **socket()** zwraca deskryptor gniazda odpowiadający punktowi końcowemu. Instrukcja ta informuje również, że dla tego gniazda zostanie użyta rodzina adresów INET (Internet Protocol) z transportem UDP transport (SOCK_DGRAM).
2. Po utworzeniu deskryptora gniazda funkcja **bind()** pobiera unikalną nazwę gniazda. W tym przykładzie numer portu nie jest określony, ponieważ aplikacja kliencka nie nawiązuje połączenia z gniazdem. Tego fragmentu kodu można użyć w innych serwerach, które korzystają z blokowania takich funkcji API, jak **accept()**.
3. Funkcja **listen()** wskazuje gotowość do zaakceptowania wysyłanych przez klienta żądań połączenia. Po wywołaniu funkcji **listen()** alarm jest ustawiany na uruchomienie po pięciu sekundach. Ten alarm lub sygnał poinformuje o zablokowaniu wywołania funkcji **accept()**.
4. Funkcja **accept()** akceptuje żądanie połączenia od klienta. Normalnie wywołanie to blokowałoby przez czas nieograniczony, ale alarm powoduje ograniczenie tego czasu do 5 sekund. Kiedy alarm się włączy, funkcja accept zakończy się z kodem powrotu -1 i wartością errno równą EINTR.
5. Funkcja **close()** zamyka wszystkie otwarte deskryptory gniazd.

Informacje dotyczące wykorzystania przykładowych kodów zawiera sekcja Informacje dotyczące kodu.

```
/******************************************************************/
/* Przykład ustawienia alarmów dla blokujących funkcji API gniazd */
/******************************************************************/

/******************************************************************/
/* Włączane pliki                                                 */
/******************************************************************/
#include <signal.h>
#include <unistd.h>
```

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

/*****************************************************************/
/* Procedura przechwytująca sygnał. Będzie ona wywoływana, kiedy  */
/* pojawi się sygnał.                                            */
/*****************************************************************/
void catcher(int sig)
{
   printf("Wywołanie procedury przechwytującej dla sygnału %d\n", sig);
}

/*****************************************************************/
/* Program główny                                                */
/*****************************************************************/
int main (int argc, char *argv[])
{
   struct sigaction sact;
   struct sockaddr_in   addr;
   time_t t;
   int sd, rc;

/*****************************************************************/
/* Utworzenie gniazda AF_INET, SOCK_STREAM                       */
/*****************************************************************/
   printf("Tworzenie gniazda TCP\n");
   sd = socket(AF_INET, SOCK_STREAM, 0);
   if (sd == -1)
   {
      perror("utworzenie gniazda nie powiodło się");
      return(-1);
   }

/*****************************************************************/
/* Powiązanie gniazda. Nie podano numeru portu, ponieważ         */
/* z gniazdem tym nie będzie nawiązywane połączenie.             */
/*****************************************************************/
   memset(&addr, 0, sizeof(addr));
   addr.sin_family = AF_INET;
   printf("Wiązanie gniazda\n");
   rc = bind(sd, (struct sockaddr *)&addr, sizeof(addr));
   if (rc != 0)
   {
      perror("powiązanie nie powiodło się");
      close(sd);
      return(-2);
   }

/*****************************************************************/
/* Wykonanie czynności nasłuchiwania przez gniazdo.              */
/*****************************************************************/
   printf("Ustawienie parametru backlog nasłuchiwania\n");
   rc = listen(sd, 5);
   if (rc != 0)
   {
      perror("nasłuchiwanie nie powiodło się");
      close(sd);
      return(-3);
   }

/*****************************************************************/
/* Ustawienie alarmu, który włączy się po pięciu sekundach.      */
/*****************************************************************/
   printf("\nUstawienie alarmu, który włączy się po 5 sek. Alarm spowoduje,\n");
```

```
   printf("że zablokowana funkcja accept() zwróci -1 i wartość errno równą EINTR.\n\n");
   sigemptyset(&sact.sa_mask);
   sact.sa_flags = 0;
   sact.sa_handler = catcher;
   sigaction(SIGALRM, &sact, NULL);
   alarm(5);

/*****************************************************************/
/* Wyświetla bieżący czas z chwili ustawienia alarmu            */
/*****************************************************************/
   time(&t);
   printf("Przed funkcją accept() godzina %s", ctime(&t));

/*****************************************************************/
/* Akceptowanie wywołania. Normalnie wywołanie to blokowałoby     */
/* przez czas nieograniczony, ale alarm powoduje ograniczenie     */
/* blokowania do 5 s. Kiedy alarm się włączy, funkcja accept      */
/* zakończy się z -1 i wartością errno równą EINTR.               */
/*****************************************************************/
   errno = 0;
   printf("Oczekiwanie na połączenie przychodzące\n");
   rc = accept(sd, NULL, NULL);
   printf("Funkcja accept() zakończona.  rc = %d, errno = %d\n", rc, errno);
   if (rc >= 0)
   {
      printf("Odebrano połączenie przychodzące\n");
      close(rc);
   }
   else
   {
      perror("łańcuch errno");
   }

/*****************************************************************/
/* Wyświetlenie godziny włączenia alarmu                         */
/*****************************************************************/
   time(&t);
   printf("Po funkcji accept(), godzina %s\n", ctime(&t));
   close(sd);
   return(0);
}
```

Przykłady: użycie rozsyłania grupowego

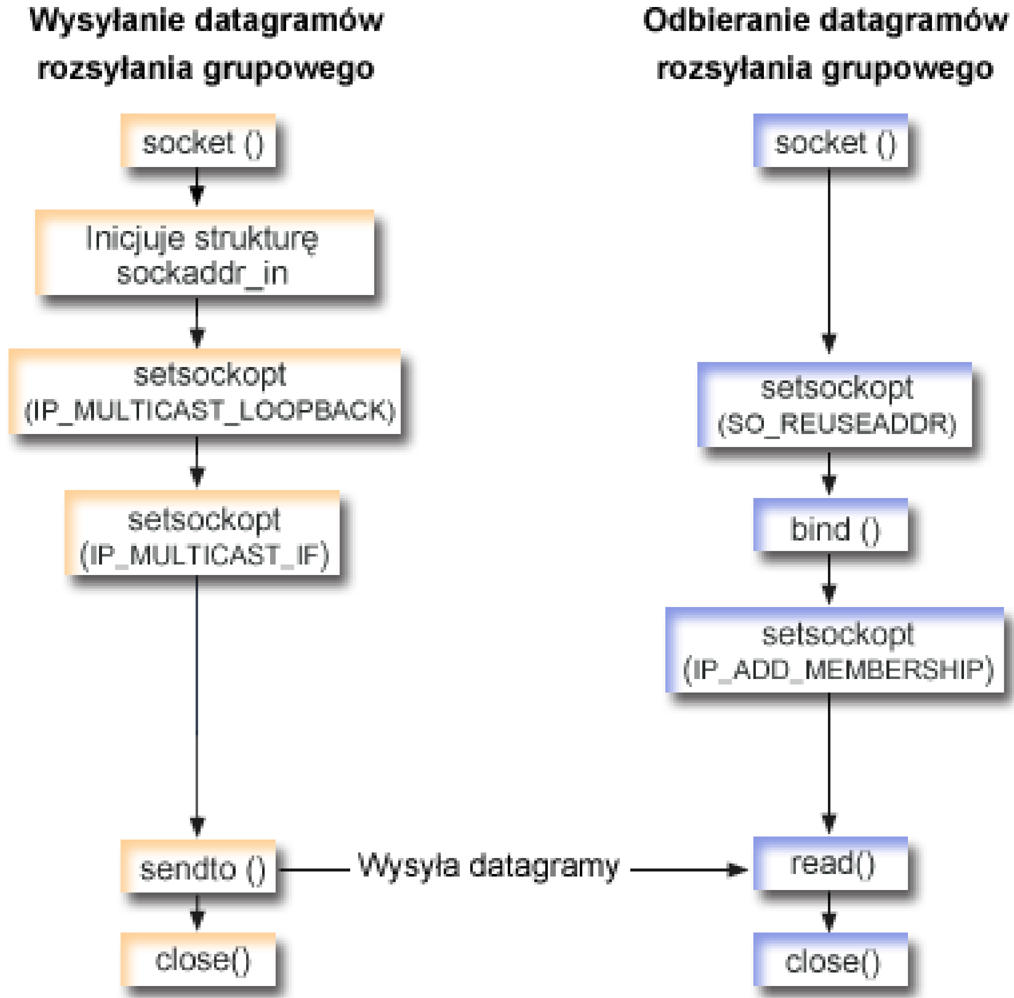
Rozsyłanie grupowe IP zapewnia aplikacjom możliwość wysyłania jednego datagramu IP, który odbierany jest przez grupę hostów w sieci. Hosty należące do grupy mogą znajdować się w tej samej podsieci lub w różnych podsieciach połączonych routerami obsługującymi rozsyłanie grupowe. Hosty mogą być dołączane do grup i usuwane z nich w każdej chwili. Nie ma ograniczeń dotyczących położenia ani liczby członków grupy hostów. Grupę hostów identyfikuje klasa D adresów internetowych w zakresie od 224.0.0.1 do 239.255.255.255.

Aplikacja może wysyłać lub odbierać rozsyłane grupowo datagramy za pomocą funkcji API **socket()** i bezpołączeniowych gniazd typu SOCK_DGRAM. Rozsyłanie grupowe jest metodą transmisji typu jeden-do-wielu. Do rozsyłania grupowego nie można używać zorientowanych na połączenie gniazd typu SOCK_STREAM. Gdy utworzone zostanie gniazdo typu SOCK_DGRAM, aplikacja może użyć funkcji **setsockopt()** w celu sterowania charakterystyką rozsyłania grupowego przypisaną do tego gniazda. Funkcja setsockopt() akceptuje następujące opcje poziomu IPPROTO_IP:

- IP_ADD_MEMBERSHIP: Umożliwia dołączenie do podanej grupy rozsyłania grupowego.
- IP_DROP_MEMBERSHIP: Umożliwia usunięcie z podanej grupy rozsyłania grupowego.
- IP_MULTICAST_IF: Konfiguruje interfejs, poprzez który wysyłane są wychodzące datagramy rozsyłania grupowego.
- IP_MULTICAST_TTL: Ustawia wartość Time To Live (TTL) w nagłówku IP wychodzącego datagramu rozsyłania grupowego.

- IP_MULTICAST_LOOP: Określa, czy kopia wychodzącego datagramu rozsyłania grupowego ma być dostarczana do hosta wysyłającego tak długo, dopóki jest on członkiem grupy rozsyłania grupowego.

Uwaga: Gniazda systemu OS/400 obsługują rozsyłanie grupowe IP dla rodziny adresów AF_INET.



Przebieg zdarzeń w gnieździe: wysyłanie datagramów rozsyłania grupowego
Poniżej wyjaśniono sekwencję wywołań funkcji gniazd, przedstawionych na powyższej ilustracji. Opisano także relacje pomiędzy dwiema aplikacjami, które wysyłają i odbierają datagramy rozsyłania grupowego. Każdy zbiór wywołań zawiera odsyłacze do uwag dotyczących użycia poszczególnych funkcji API. Aby uzyskać szczegółowe informacje dotyczące użycia tych funkcji API, należy użyć poniższych odsyłaczy. W sekcji zawierającej program wysyłający datagramy rozsyłania grupowego użyto wywołań następujących funkcji:

1. Funkcja **socket()** zwraca deskryptor gniazda odpowiadający punktowi końcowemu. Instrukcja ta informuje również, że dla tego gniazda zostanie użyta rodzina adresów INET (Internet Protocol) z transportem TCP transport (SOCK_DGRAM). Gniazdo to będzie wysyłało datagramy do drugiej aplikacji.
2. Struktura sockaddr_in określa docelowy adres IP i numer portu. W tym przykładzie adresem jest 225.1.1.1, a portem - 5555.
3. Funkcja **setsockopt()** ustawia opcję gniazda IP_MULTICAST_LOOP, aby system wysyłający nie otrzymywał wysłanych przezeń datagramów rozsyłania grupowego.

4. Funkcja **setsockopt()** używa opcji gniazd IP_MULTICAST_IF, która definiuje interfejs lokalny, przez który będą przesyłane datagramy rozsyłania grupowego.
5. Funkcja **sendto()** wysyła datagramy rozsyłania grupowego do podanych grupowych adresów IP.
6. Funkcja **close()** zamyka wszystkie otwarte deskryptory gniazd.

Przebieg zdarzeń w gnieździe: odbieranie datagramów rozsyłania grupowego
W sekcji zawierającej program odbierający datagramy rozsyłania grupowego użyto wywołań następujących funkcji:

1. Funkcja **socket()** zwraca deskryptor gniazda odpowiadający punktowi końcowemu. Instrukcja ta informuje również, że dla tego gniazda zostanie użyta rodzina adresów INET (Internet Protocol) z transportem TCP transport (SOCK_DGRAM). Gniazdo to będzie wysyłało datagramy do drugiej aplikacji.
2. Funkcja **setsockopt()** ustawia opcję gniazd SO_REUSEADDR, umożliwiającą wielu aplikacjom odbieranie datagramów skierowanych do portu lokalnego o tym samym numerze.
3. Funkcja **bind()** określa numer portu lokalnego. W tym przykładzie adres IP jest podany w postaci INADDR_ANY, aby możliwe było odbieranie datagramów przeznaczonych dla grupy.
4. Funkcja **setsockopt()** używa opcji gniazd IP_ADD_MEMBERSHIP, łączącej grupę, dla której datagramy są przeznaczone. Dołączając do grupy należy podać adres grupy klasy D razem z adresem IP lokalnego interfejsu. System musi wywołać opcję gniazda IP_ADD_MEMBERSHIP dla każdego lokalnego interfejsu, który odbiera datagramy rozsyłania grupowego. W tym przykładzie grupa rozsyłania (225.1.1.1) jest połączona z interfejsem lokalnym 9.5.1.1.

   Uwaga: Opcję IP_ADD_MEMBERSHIP należy wywołać dla każdego interfejsu lokalnego, przez który datagramy rozsyłania grupowego mają być odbierane.
5. Funkcja **read()** odczytuje wysłane datagramy rozsyłania grupowego.
6. Funkcja **close()** zamyka wszystkie otwarte deskryptory gniazd.

Przykład: wysyłanie datagramów rozsyłania grupowego

Poniższy przykład umożliwia przeprowadzenie dla gniazda następujących kroków służących do wysłania datagramów rozsyłania grupowego. Opis przebiegu zdarzeń w gnieździe dla tego programu znajduje się w sekcji Przykłady: użycie rozsyłania grupowego.

Informacje dotyczące wykorzystania przykładowych kodów zawiera sekcja Informacje dotyczące kodu.

```
#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[])
{

  /* ------------------------------------------------------------*/
  /*                                                             */
  /* Przykład kodu SMD (Wysłanie datagramu rozsyłania grupowego) */
  /*                                                             */
  /* ------------------------------------------------------------*/

  /*
   * Utworzenie gniazda datagramu, do którego  *
   * datagram ma być wysłany.
   */
```

```
  sd = socket(AF_INET, SOCK_DGRAM, 0);
  if (sd < 0) {
    perror("otwieranie gniazda datagramowego");
    exit(1);
  }

  /*
   * Inicjowanie struktury grupy sockaddr z
   * adresem grupy 225.1.1.1 i portem 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);

  /*
   * Wyłączenie pętli zwrotnej, aby nie otrzymywać własnych datagramów.
   */
  {
    char loopch=0;

    if (setsockopt(sd, IPPROTO_IP, IP_MULTICAST_LOOP,
                   (char *)&loopch, sizeof(loopch)) < 0) {
      perror("ustawienie IP_MULTICAST_LOOP:");
      close(sd);
      exit(1);
    }
  }

  /*
   * Skonfigurowanie lokalnego interfejsu dla wychodzących datagramów
   * rozsyłania grupowego.
   * Podany adres IP musi być przypisany do lokalnego interfejsu,
   * który obsługuje rozsyłanie grupowe.
   */
  localInterface.s_addr = inet_addr("9.5.1.1");
  if (setsockopt(sd, IPPROTO_IP, IP_MULTICAST_IF,
                 (char *)&localInterface,
                 sizeof(localInterface)) < 0) {
    perror("konfigurowanie lokalnego interfejsu");
    exit(1);
  }


  /*
   * Wysłanie komunikatu do grupy rozsyłania grupowego podanej przez
   * strukturę groupSock sockaddr.
   */
  datalen = 10;
  if (sendto(sd, databuf, datalen, 0,
             (struct sockaddr*)&groupSock,
             sizeof(groupSock)) < 0)
  {
    perror("wysłanie komunikatu datagramu");
  }
}
```

Przykład: odbieranie datagramów rozsyłania grupowego

Poniższy przykład umożliwia przeprowadzenie dla gniazda następujących kroków służących do odbierania datagramów rozsyłania grupowego. Opis przebiegu zdarzeń w gnieździe dla tego programu znajduje się w sekcji Przykłady: użycie rozsyłania grupowego.

Informacje dotyczące wykorzystania przykładowych kodów zawiera sekcja Informacje dotyczące kodu.

```
#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[])
{

  /* ------------------------------------------------------------*/
  /*                                                             */
  /* Przykład kodu RMD (Odebranie datagramu rozsyłania grupowego)*/
  /*                                                             */
  /* ------------------------------------------------------------*/

  /*
   * Utworzenie gniazda, z którego  *
   * datagram ma być odebrany.
   */
  sd = socket(AF_INET, SOCK_DGRAM, 0);
  if (sd < 0) {
    perror("otwieranie gniazda datagramowego");
    exit(1);
  }

  /*
   * Włącz SO_REUSEADDR, aby umożliwić wielu instancjom tej aplikacji
   * odbieranie kopii datagramów rozsyłania grupowego.
   */
  {
    int reuse=1;

    if (setsockopt(sd, SOL_SOCKET, SO_REUSEADDR,
                   (char *)&reuse, sizeof(reuse)) < 0) {
      perror("ustawianie SO_REUSEADDR");
      close(sd);
      exit(1);
    }
  }

  /*
   * Powiąż odpowiedni numer portu z adresem IP
   * podanym jako 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("wiązanie gniazda datagramu");
    close(sd);
    exit(1);
  }


  /*
   * Dołącz do grupy rozsyłania grupowego 225.1.1.1 w lokalnym
```

```
   * interfejsie
   * 9.5.1.1. Zauważ, że opcja IP_ADD_MEMBERSHIP musi być wywołana
   * dla każdego interfejsu lokalnego, przez który datagramy
   * rozsyłania grupowego mają być odbierane.
   */
  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("dodawanie grupy rozsyłania grupowego");
    close(sd);
    exit(1);
  }

  /*
   * Odczyt z gniazda.
   */
  datalen = sizeof(databuf);
  if (read(sd, databuf, datalen) < 0) {
    perror("odczyt datagramu z komunikatem");
    close(sd);
    exit(1);
  }

}
```

Przykład: odpytywanie i aktualizacja serwera DNS

Poniższy przykład pokazuje, jak wysyłać zapytania do systemu nazw domen (DNS) i jak aktualizować jego rekordy.

Informacje dotyczące wykorzystania przykładowych kodów zawiera sekcja Informacje dotyczące kodu.

```
/**************************************************************************/
/* Ten program aktualizuje DNS przy użyciu sygnatury transakcji (TSIG),   */
/* służącej do podpisywania pakietów aktualizacyjnych. Następnie odpytuje */
/* serwer DNS, aby sprawdzić powodzenie aktualizacji.                     */
/**************************************************************************/

/**************************************************************************/
/* Wymagane przez program pliki nagłówkowe.                               */
/**************************************************************************/
#include <stdio.h>
#include <errno.h>
#include <arpa/inet.h>
#include <resolv.h>
#include <netdb.h>

/**************************************************************************/
/* Deklaracja rekordów aktualizacji - rekordu strefy, rekordu wstępnego   */
/* i 2 rekordów aktualizacji.                                             */
/**************************************************************************/
ns_updrec update_records[] =
{
   {
      {NULL,&update_records[1]},
      {NULL,&update_records[1]},
      ns_s_zn,                 /* rekord strefy */
      "mydomain.ibm.com.",
      ns_c_in,
      ns_t_soa,
      0,
      NULL,
      0,
      0,
      NULL,
      NULL,
      0
```

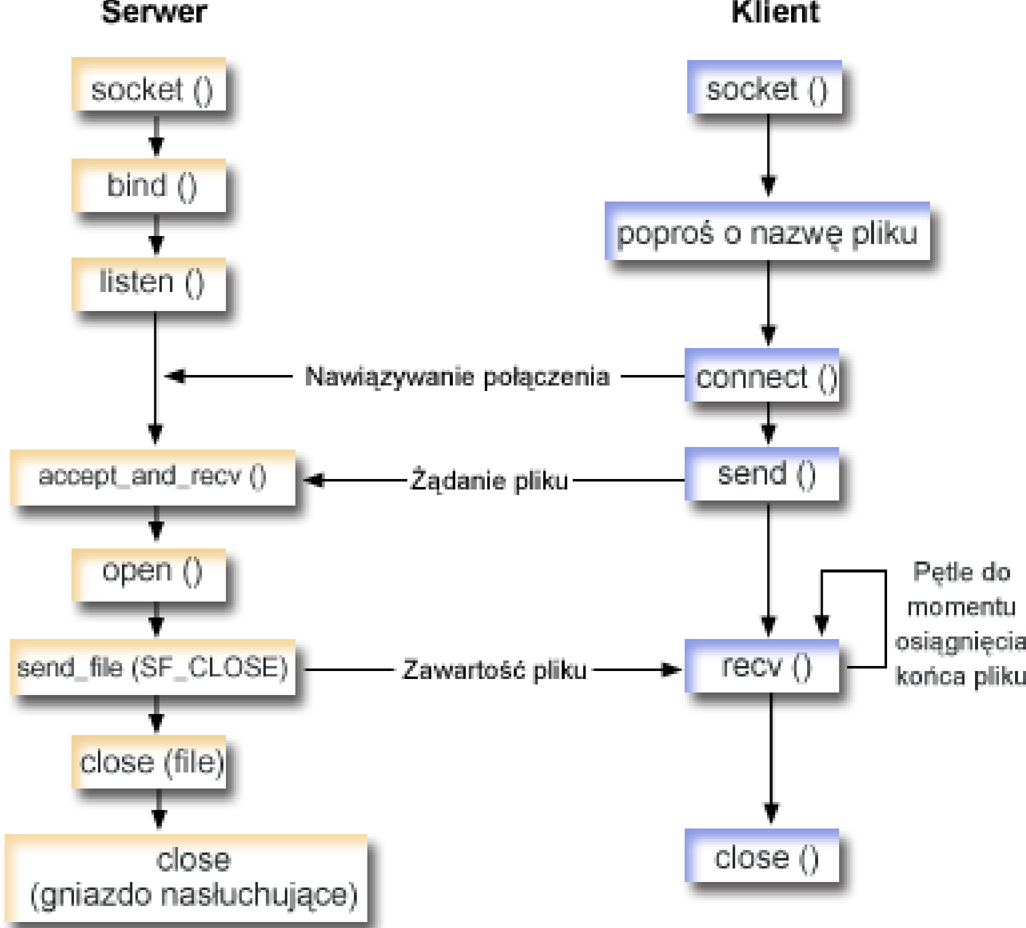
```
   },
   {
      {&update_records[0],&update_records[2]},
      {&update_records[0],&update_records[2]},
      ns_s_pr,                /* rekord wstępny */
      "mypc.mydomain.ibm.com.",
      ns_c_in,
      ns_t_a,
      0,
      NULL,
      0,
      ns_r_nxdomain,    /* rekord nie może istnieć */
      NULL,
      NULL,
      0
   },
   {
      {&update_records[1],&update_records[3]},
      {&update_records[1],&update_records[3]},
      ns_s_ud,                 /* rekord aktualizujący */
      "mypc.mydomain.ibm.com.",
      ns_c_in,
      ns_t_a,                  /* adres IPv4... */
      10,
      (unsigned char *)"10.10.10.10",
      11,
      ns_uop_add,              /* ...który ma zostać dodany */
      NULL,
      NULL,
      0
   },
   {
      {&update_records[2],NULL},
      {&update_records[2],NULL},
      ns_s_ud,                 /* rekord aktualizujący */
      "mypc.mydomain.ibm.com.",
      ns_c_in,
      ns_t_aaaa,               /* adres IPv4... */
      10,
      (unsigned char *)"fedc:ba98:7654:3210:fedc:ba98:7654:3210",
      39,
      ns_uop_add,              /* ...który ma zostać dodany */
      NULL,
      NULL,
      0
   }
};

/***************************************************************************/
/* Dwie poniższe struktury definiują klucz i klucz tajny, które muszą     */
/* być zgodne ze skonfigurowanymi w serwerze DNS:                         */
/*   allow-update {                                                        */
/*   key my-long-key.;                                                     */
/*   }                                                                     */
/*                                                                         */
/* Musi to być binarny równoważnik tajnego klucza base64                  */
/*                                                                         */
/***************************************************************************/
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",            /* Ten klucz musi istnieć na serwerze DNS */
   NS_TSIG_ALG_HMAC_MD5,
```

```
   secret,
   sizeof(secret)
};

void main()
{
   /*******************************************************************/
   /* Definicje zmiennych i struktur.                                 */
   /*******************************************************************/
   struct state res;
   int result, update_size;
   unsigned char update_buffer[2048];
   unsigned char answer_buffer[2048];
   int buffer_length = sizeof(update_buffer);

   /* Wyłączenie opcji init w celu zainicjowania struktury */
   res.options &= ~ (RES_INIT | RES_XINIT);

   result = res_ninit(&res);

   /* Umieszczenie przetwarzania w tym miejscu pozwoli sprawdzić wyniki   */
   /* i obsłużyć błędy                                                    */

   /* Budowanie bufora aktualizacji (pakietu do wysłania) na podstawie    */
   /* rekordów aktualizacji                                               */
   update_size = res_nmkupdate(&res, update_records,
                            update_buffer, buffer_length);

   /* Umieszczenie przetwarzania w tym miejscu pozwoli sprawdzić wyniki   */
   /* i obsłużyć błędy                                                    */

   {
      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;

   /* Znalezienie autorytatywnego serwera DNS dla domeny, która ma być    */
   /* aktualizowana                                                       */

      result = res_findzonecut(&res, "mypc.mydomain.ibm.com", ns_c_in, 0,
      zone_name, zone_name_size,
                            addresses, number_addresses);

   /* Umieszczenie przetwarzania w tym miejscu pozwoli sprawdzić wyniki   */
   /* i obsłużyć błędy                                                    */

   /* Sprawdzenie, czy znaleziony serwer DNS jest jednym z używanych      */
      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);

   /* Umieszczenie przetwarzania w tym miejscu pozwoli sprawdzić wyniki   */
   /* i obsłużyć błędy                                                    */

   /* Ustawienie adresu DNS znalezionego jako res_findzonecut w strukturze    */
   /* res. Do tego serwera DNS zostanie wysłana aktualizacja z podpisem TSIG. */
      res.nscount = 1;
      res.nsaddr_list[0] = s_address;

   /* Wysłanie aktualizacji DNS z podpisem TSIG                           */
      result = res_nsendsigned(&res, update_buffer, update_size,
                            &my_key,
```

```
                               answer_buffer, sizeof answer_buffer);

   /* Umieszczenie przetwarzania w tym miejscu pozwoli sprawdzić wyniki   */
   /* i obsłużyć błędy                                                    */
   }

   /*********************************************************************/
   /* res_findzonecut(), res_nmkupdate(), i res_nsendsigned() można      */
   /* zastąpić jednym wywołaniem res_nupdate() z użyciem                 */
   /* update_records[1] w celu pominięcia rekordu strefy:                */
   /*                                                                   */
   /* result = res_nupdate(&res, &update_records[1], &my_key);          */
   /*                                                                   */
   /*********************************************************************/
   /*********************************************************************/
   /* Sprawdzenie, czy aktualizacja rzeczywiście powiodła się!          */
   /* Wybrano protokół TCP, a nie UDP, więc po zainicjowaniu zmiennej res */
   /* należy ustawić odpowiednią opcję. Ponadto program będzie ignorował  */
   /* lokalną pamięć podręczną i zawsze wysyłał zapytania do serwera DNS. */
   /*********************************************************************/

   res.options |= RES_USEVC|RES_NOCACHE;

   /* Wysłanie zapytania o rekordy adresu mypc.mydomain.ibm.com           */
   result = res_nquerydomain(&res,"mypc", "mydomain.ibm.com.",
                             ns_c_in, ns_t_a,
                             update_buffer, buffer_length);

   /* Przykład obsługi błędów i drukowania komunikatów o błędach         */
   if (result == -1)
   {
      printf("\nZapytanie nie powiodło się.  Wynik = %d \nerrno: %d: %s \
              \nh_errno: %d: %s",
             result,
            errno, strerror(errno),
             h_errno, hstrerror(h_errno));
   }
   /*********************************************************************/
   /* Komunikat o błędzie będzie wyglądał następująco:                  */
   /*                                                                   */
   /* zapytanie o domenę nie powiodło się. Wynik = -1                   */
   /* errno: 0: Brak błędu.                                             */
   /* h_errno: 5: Nieznany host                                         */
   /*********************************************************************/
   return;
}
```

Przykład: przesyłanie danych za pomocą funkcji send_file() i accept_and_recv()

Poniższe przykłady umożliwiają serwerowi komunikację z klientem za pomocą funkcji **send_file()** i **accept_and_recv()**.



Przebieg zdarzeń w gnieździe: serwer wysyła zawartość pliku
Poniżej wyjaśniono sekwencję wywołań funkcji gniazd, przedstawionych na powyższej ilustracji. Opisano także relacje pomiędzy dwiema aplikacjami, które wysyłają i odbierają pliki. Każdy zbiór wywołań zawiera odsyłacze do uwag dotyczących użycia poszczególnych funkcji API. Aby uzyskać szczegółowe informacje dotyczące użycia tych funkcji API, należy użyć poniższych odsyłaczy. W sekcji Przykład: użycie funkcji accept_and_recv() i send_file() do wysyłania zawartości pliku użyto następujących wywołań funkcji:

1. Serwer tworzy gniazdo nasłuchujące, wywołując funkcje **socket()**, **bind()** i **listen()**.
2. Serwer inicjuje struktury adresu lokalnego i zdalnego.
3. Serwer wywołuje funkcję **accept_and_recv()** umożliwiającą oczekiwanie na połączenie przychodzące i pierwszy bufor danych przekazany poprzez to połączenie. Wywołanie to zwraca liczbę odebranych bajtów oraz adres lokalny i zdalny przypisane do tego połączenia. Jest ono kombinacją funkcji **accept()**, **getsockname()** i **recv()**.
4. Serwer otwiera plik, którego nazwa została uzyskana od aplikacji klienta jako dane z funkcją **accept_and_recv()** poprzez wywołanie funkcji **open()**.
5. Funkcja **memset()** zeruje wszystkie pola struktury sf_parms. Serwer wstawia w polu deskryptora pliku wartość zwróconą przez funkcję **open()**. Następnie serwer podaje wartość pola wielkości pliku wynoszącą -1, co wskazuje, że serwer powinien przesłać cały plik. System wysyła cały plik, więc nie trzeba podawać wartości pola pozycji w pliku.
6. Serwer przesyła zawartość pliku za pomocą funkcji **send_file()**. Funkcja **send_file()** nie kończy działania, dopóki cały plik nie zostanie wysłany lub funkcja zostanie przerwana. Funkcja **send_file()** jest efektywniejsza, ponieważ aplikacja nie musi wchodzić w pętlę funkcji **read()** i **send()**, dopóki plik się nie skończy.

7. Serwer podaje opcję SF_CLOSE dla funkcji API **send_file()**. Flaga SF_CLOSE informuje funkcję **send_file()** o konieczności automatycznego zamknięcia połączenia gniazda z chwilą pomyślnego przesłania ostatniego bajtu pliku i bufora końcowego (jeśli został podany). Aplikacja nie musi wywoływać funkcji **close()**, jeśli podana została opcja SF_CLOSE.

Przebieg zdarzeń w gnieździe: klient żąda pliku
W sekcji Przykład: klient żądający pliku użyto wywołań następujących funkcji:

1. Klient pobiera nie więcej niż dwa parametry.

   Pierwszy parametr (jeśli podany) jest adresem IP w postaci dziesiętnej z kropkami lub nazwą hosta, w którym znajduje się aplikacja serwera.

   Drugi parametr (jeśli podany) jest nazwą pliku, który klient próbuje pobrać z serwera. Aplikacja serwera wysyła do klienta zawartość podanego pliku. Jeśli użytkownik nie poda żadnego parametru, klient jako adresu IP serwera użyje parametru INADDR_ANY. Jeśli użytkownik nie poda drugiego parametru, program zażąda wpisania nazwy pliku.
2. Wywołanie przez klienta funkcji **socket()** w celu utworzenia deskryptora gniazda.
3. Wywołanie przez klienta funkcji **connect()** w celu nawiązania połączenia z serwerem. Adres IP serwera uzyskano w pierwszym kroku.
4. Wywołanie przez klienta funkcji **send()** w celu podania serwerowi nazwy pliku, który ma być przesłany. Nazwę tę uzyskano w pierwszym kroku.
5. Klient przechodzi do pętli ″do″ wywołującej funkcję **recv()** do czasu, aż przetwarzanie osiągnie koniec pliku. Kod powrotu 0 w funkcji **recv()** oznacza, że serwer zamknął połączenie.
6. Wywołanie przez klienta funkcji **close()** w celu zamknięcia gniazda.

Przykład: użycie funkcji accept_and_recv() i send_file() do wysyłania zawartości pliku

W poniższym przykładzie serwer wykonuje wymienione niżej czynności w celu nawiązania komunikacji za pomocą funkcji **send_file()** i **accept_and_recv()**.

Informacje dotyczące wykorzystania przykładowych kodów zawiera sekcja Informacje dotyczące kodu.

```
/*************************************************/
/* Przykładowy serwer wysyłający dane do klienta */
/*************************************************/

#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];

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

```
/* Jeśli podano argument, użyj go do sterowania  */
/* liczbą połączeń przychodzących                */
/*************************************************/
if (argc >= 2)
   num = atoi(argv[1]);
else
   num = 1;

/*************************************************/
/* Utwórz gniazdo strumienia AF_INET do          */
/* odbierania połączeń przychodzących            */
/*************************************************/
listen_sd = socket(AF_INET, SOCK_STREAM, 0);
if (listen_sd < 0)
{
   perror("Niepowodzenie socket()");
   exit(-1);
}

/*************************************************/
/* Ustaw bit SO_REUSEADDR, aby nie trzeba było   */
/* czekać 2 minuty zanim serwer zostanie ponownie*/
/* uruchomiony                                   */
/*************************************************/
rc = setsockopt(listen_sd,
                SOL_SOCKET,
                SO_REUSEADDR,
                (char *)&flag,
                sizeof(flag));
if (rc < 0)
{
   perror("Niepowodzenie accept()");
   close(listen_sd);
   exit(-1);
}

/*************************************************/
/* Powiąż gniazdo                                */
/*************************************************/
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("Niepowodzenie bind()");
   close(listen_sd);
   exit(-1);
}

/*************************************************/
/* Ustaw parametr backlog funkcji listen         */
/*************************************************/
rc = listen(listen_sd, 5);
if (rc < 0)
{
   perror("Niepowodzenie listen()");
   close(listen_sd);
   exit(-1);
}

/*************************************************/
/* Zainicjowanie długości adresu lokalnego       */
/* i zdalnego                                    */
/*************************************************/
```

```
local_addr_length  = sizeof(local_addr);
remote_addr_length = sizeof(remote_addr);

/*************************************************/
/* Poinformuj użytkownika o tym, że             */
/* serwer jest gotowy                            */
/*************************************************/
printf("Serwer jest gotowy\n");

/*************************************************/
/* Przejdź przez pętlę raz dla każdego połączenia*/
/*************************************************/
for (i=0; i < num; i++)
{
   /**********************************************/
   /* Oczekiwanie na połączenie przychodzące     */
   /**********************************************/
   printf("Iteracja: %d\n", i+1);
   printf("  oczekiwanie na 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("Niepowodzenie accept_and_recv()");
      close(listen_sd);
      close(accept_sd);
      exit(-1);
   }
   printf("  Żądanie pliku: %s\n", buffer);

   /**********************************************/
   /* Otwórz plik do pobrania                    */
   /**********************************************/
   fd = open(buffer, O_RDONLY);
   if (fd < 0)
   {
      perror("Niepowodzenie open()");
      close(listen_sd);
      close(accept_sd);
      exit(-1);
   }

   /**********************************************/
   /* Zainicjowanie struktury sf_parms           */
   /**********************************************/
   memset(&parms, 0, sizeof(parms));
   parms.file_descriptor = fd;
   parms.file_bytes      = -1;

   /**********************************************/
   /* Zainicjowanie licznika całkowitej liczby   */
   /* przesłanych bajtów                         */
   /**********************************************/
   total_sent = 0;

   /**********************************************/
   /* Pętla, aż do przesłania całego pliku       */
   /**********************************************/
   do
   {
```

```
         rc = send_file(&accept_sd, &parms, SF_CLOSE);
         if (rc < 0)
         {
            perror("Niepowodzenie send_file()");
            close(fd);
            close(listen_sd);
            close(accept_sd);
            exit(-1);
         }
         total_sent += parms.bytes_sent;

      } while (rc == 1);

      printf("Całkowita liczba przesłanych bajtów: %d\n, total_sent);

      /*************************************************/
      /* Zamknij wysłany plik                          */
      /*************************************************/
      close(fd);
   }

   /****************************************************/
   /* Zamknięcie gniazda nasłuchującego                */
   /****************************************************/
   close(listen_sd);

   /****************************************************/
   /* Zamknij gniazdo akceptujące                      */
   /****************************************************/
   if (accept_sd != -1)
      close(accept_sd);
}
```

Przykład: klient żądający pliku

Poniższy przykład umożliwia programowi klienta zażądanie pliku z serwera i zaczekanie, aż serwer w odpowiedzi odeśle zawartość pliku.

Informacje dotyczące wykorzystania przykładowych kodów zawiera sekcja Informacje dotyczące kodu.

```
/**************************************************/
/* Przykładowy klient żądający pliku od serwera  */
/**************************************************/
#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;

   /****************************************************/
   /* Inicjowanie struktury adresów gniazd             */
   /****************************************************/
   memset(&addr, 0, sizeof(addr));
   addr.sin_family = AF_INET;
```

```
addr.sin_port   = htons(SERVER_PORT);

/*************************************************/
/* Określ nazwę hosta i adres IP maszyny, na     */
/* której działa serwer                          */
/*************************************************/
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 nie znaleziony!\n");
      exit(-1);
   }
   memcpy((char *)&addr.sin_addr.s_addr,
          host_ent->h_addr_list[0],
          host_ent->h_length);
}

/***************************************************/
/* Sprawdź, czy użytkownik podał w wierszu komend */
/* nazwę pliku                                     */
/***************************************************/
if (argc == 3)
{
   strcpy(filename, argv[2]);
}
else
{
   printf("Wpisz nazwę pliku:\n");
   gets(filename);
}

/*************************************************/
/* Utwórz gniazdo strumienia AF_INET             */
/*************************************************/
sockfd = socket(AF_INET, SOCK_STREAM, 0);
if (sockfd < 0)
{
   perror("Niepowodzenie socket()");
   exit(-1);
}
printf("Gniazdo zakończyło działanie.\n");

/*************************************************/
/* Połączenie z serwerem                         */
/*************************************************/
rc = connect(sockfd,
             (struct sockaddr *)&addr,
             sizeof(struct sockaddr_in));
if (rc < 0)
{
   perror("Niepowodzenie connect()");
   close(sockfd);
   exit(-1);
}
printf("Połączenie nawiązane.\n");

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

```
   /* Wysłanie żądania do serwera                  */
   /************************************************/
   rc = send(sockfd, filename, strlen(filename) + 1, 0);
   if (rc < 0)
   {
      perror("Niepowodzenie send()");
      close(sockfd);
      exit(-1);
   }
   printf("Wysłano żądanie %s\n", filename);

   /************************************************/
   /* Odebranie pliku z serwera                    */
   /************************************************/
   do
   {
      rc = recv(sockfd, buffer, sizeof(buffer), 0);
      if (rc < 0)
      {
         perror("Niepowodzenie recv()");
         close(sockfd);
         exit(-1);
      }
      else if (rc == 0)
      {
         printf("Koniec pliku\n");
         break;
      }
      printf("Otrzymano bajtów: %d\n", rc);

   } while (rc > 0);

   /************************************************/
   /* Zamknij gniazdo                              */
   /************************************************/
   close(sockfd);
}
```

Narzędzie Xsockets

Narzędzie Xsockets jest jednym z wielu narzędzi dostarczanych z systemem iSeries. Wszystkie narzędzia znajdują się w bibliotece QUSRTOOL. Xsockets umożliwia programistom interaktywną pracę z funkcjami API gniazd. Narzędzie Xsockets umożliwia realizację następujących zadań:

- zapoznanie się z funkcjami API gniazd,
- interaktywne odtworzenie konkretnych scenariuszy, co ułatwia debugowanie.

Uwaga: Narzędzie Xsockets jest dostarczane w stanie, w jakim się znajduje (″as is″).

Wymagania wstępne dla narzędzia Xsockets

- Zainstalowane środowisko programistyczne ILE C/400.
- Zainstalowany program licencjonowany 5722–SS1 ’Openness Include’.
- Zainstalowany program licencjonowany 5722–DG1 ’IBM HTTP Server’.

  Uwaga: Program ten jest potrzebny do korzystania z narzędzia Xsockets za pośrednictwem przeglądarki WWW.
- Zainstalowany program licencjonowany 5722–JV1 ’Developer Kit for Java’.

  Uwaga: Program ten jest potrzebny do korzystania z narzędzia Xsockets za pośrednictwem przeglądarki WWW.

Aby zainstalować narzędzie Xsockets i używać go, należy zapoznać się z treścią następujących sekcji:

Konfigurowanie Xsockets
W sekcji tej objaśniono, w jaki sposób utworzyć narzędzie Xsockets, które jest pomocne w projektowaniu i kompilowaniu programów używających gniazd.

Używanie Xsockets
W tej sekcji opisano, jak korzystać z narzędzia Xsockets.

Dostosowywanie Xsockets
W tej sekcji opisano, jak dostosować narzędzie Xsockets do indywidualnych wymagań.

Konfigurowanie Xsockets

Istnieją dwie wersje tego narzędzia. Pierwszą jest klient rodzimy w systemie iSeries. Wersja ta jest w całości tworzona za pomocą pierwszego zestawu instrukcji. Druga wersja jako klienta używa przeglądarki WWW. Aby móc korzystać z klienta opartego na przeglądarce WWW, należy najpierw wykonać czynności związane z konfigurowaniem wersji rodzimej.

Aby utworzyć narzędzie Xsockets, wykonaj następujące czynności:

1. Aby rozpakować narzędzie, wpisz:
   ```
   CALL QUSRTOOL/UNPACKAGE ('*ALL      ' 1)
   ```
   w wierszu komend.

   Uwaga: Pomiędzy rozpoczynającym a kończącym znakiem apostrofu (') musi być 10 znaków.
2. Aby do listy bibliotek dodać bibliotekę QUSRTOOL, wpisz
   ```
   ADDLIBLE QUSRTOOL
   ```
   w wierszu komend.
3. Utwórz bibliotekę, w której zostaną utworzone pliki programu Xsocket, wpisując
   ```
   CRTLIB <nazwa-biblioteki>
   ```
   w wierszu komend. Parametr <nazwa-biblioteki> określa bibliotekę, w której mają być utworzone obiekty narzędzia Xsockets. Na przykład:
   ```
   CRTLIB MYXSOCKET
   ```
   jest poprawną nazwą biblioteki.

   Uwaga: Obiektów narzędzia Xsocket nie należy dodawać do biblioteki QUSRTOOL. Może to powodować konflikt z innymi narzędziami w tym katalogu.
4. Aby do listy bibliotek dodać tę bibliotekę, w wierszu komend wpisz
   ```
   ADDLIBLE <nazwa-biblioteki>
   ```
   Parametr <nazwa-biblioteki> określa bibliotekę utworzoną w Kroku 3. Na przykład, aby określić bibliotekę MYXSOCKET, wpisz:
   ```
   ADDLIBLE MYXSOCKET
   ```
5. Utwórz program instalacyjny TSOCRT, który automatycznie zainstaluje narzędzie Xsockets, wpisując:
   ```
   CRTCLPGM <nazwa-biblioteki>/TSOCRT QUSRTOOL/QATTCL
   ```
   Parametr <nazwa-biblioteki> określa bibliotekę utworzoną w Kroku 3. Na przykład, aby określić bibliotekę MYXSOCKET, wpisz:
   ```
   CRTCLPGM MYXSOCKET/TSOCRT QUSRTOOL/QATTCL
   ```
6. Aby uruchomić program instalacyjny, w wierszu komend wpisz:
   ```
   CALL TSOCRT nazwa-biblioteki
   ```

W miejscu parametru nazwa-biblioteki podaj bibliotekę utworzoną w Kroku 3. Na przykład, aby określić bibliotekę MYXSOCKET, wpisz:

```
CALL TSOCRT MYXSOCKET
```

Uwaga: Program ten może zakończyć działanie po kilku minutach.

Jeśli użytkownik uruchamiający program TSOCRT nie ma uprawnienia specjalnego do zarządzania zadaniem (*JOBCTL), funkcja gniazda **givedescriptor()** zwróci błędy podczas próby przekazania deskryptora do innego zadania, niż właśnie uruchomione.

Program TSOCRT utworzy program CL, program ILE C/400 (dwa moduły), 2 programy serwisowe C/400 (dwa moduły) i trzy zbiory ekranowe. Za każdym razem, gdy to narzędzie ma być użyte, należy dodać bibliotekę do listy. Wszystkie obiekty utworzone przez narzędzie będą miały nazwę z przedrostkiem TSO.

Następnie można:

Użyć rodzimego programu Xsockets
Aby używać rodzimej wersji programu Xsockets, należy pozostać przy tym temacie. Obejmuje on podstawy użycia rodzimego narzędzia Xsockets.

Uwaga: Wersja rodzima nie obsługuje funkcji API GSKit gniazd chronionych. Aby napisać program gniazd korzystający z tych funkcji, należy użyć wersji opartej na przeglądarce WWW.

Skonfigurować Xsockets do korzystania z przeglądarki WWW

Uwaga: Czynność ta jest opcjonalna.

Co tworzy rodzima konfiguracja Xsocket

Poniższa tabela zawiera listę obiektów tworzonych przez program instalacyjny. Wszystkie obiekty będą rezydować w podanej bibliotece.

Tabela 20. Obiekty utworzone podczas instalacji narzędzia Xsockets

Nazwa obiektu	Nazwa podzbioru	Nazwa zbioru źródłowego	Typ obiektu	Rozszerzenie	Opis
TSOJNI	TSOJNI	QATTSYSC	*MODULE	C	Moduł stanowiący interfejs pomiędzy JSP a TSOSTSOC.
TSODLT	TSODLT	QATTCL	*PGM	CLP	Program CL do usuwania obiektów narzędzia i/lub podzbiorów zbioru źródłowego.
TSOXSOCK	nie dotyczy	nie dotyczy	*PGM	C	Główny program używany przez interaktywne narzędzie SOCKETS.
TSOXGJOB	nie dotyczy	nie dotyczy	*SRVPGM	C	Program serwisowy służy do obsługi interaktywnego narzędzia SOCKETS.

Tabela 20. Obiekty utworzone podczas instalacji narzędzia Xsockets (kontynuacja)

TSOJNI	nie dotyczy	nie dotyczy	*SRVPGM	C	Program serwisowy stanowiący interfejs pomiędzy JSP a TSOSTSOC, służący do obsługi interaktywnego narzędzia SOCKETS.
TSOXSOCK	TSOXSOCK	QATTSYSC	*MODULE	C	Moduł używany podczas tworzenia programu TSOXSOCK. Plik ten zawiera procedurę main().
TSOSTSOC	TSOSTSOC	QATTSYSC	*MODULE	C	Moduł używany podczas tworzenia programu TSOXSOCK. Zbiór źródłowy zawiera właściwe procedury wywołujące funkcje gniazd.
TSOXGJOB	TSOXGJOB	QATTSYSC	*MODULE	C	Moduł używany podczas tworzenia programu TSOXGJOB. Zbiór źródłowy zawiera procedurę identyfikującą zadanie wewnętrzne. Ten wewnętrzny identyfikator zadania składa się z nazwy zadania, ID użytkownika i numeru zadania.
TSODSP	TDSPDSP	QATTDDS	*FILE	DSPF	Zbiór ekranowy używany przez narzędzie Xsockets do tworzenia głównego ekranu zawierającego funkcje gniazd.
TSOFUN	TDSOFUN	QATTDDS	*FILE	DSPF	Zbiór ekranowy używany przez narzędzie Xsockets do obsługi różnych funkcji gniazd.
TSOMNU	TDSOMNU	QATTDDS	*FILE	DSPF	Zbiór ekranowy używany przez narzędzie Xsockets do obsługi paska menu.

Tabela 20. Obiekty utworzone podczas instalacji narzędzia Xsockets (kontynuacja)

QATTIFS2	nie dotyczy	nie dotyczy	*FILE	PF-DTA	Zawiera plik JAR używany przez serwer Tomcat.

Konfigurowanie Xsockets do korzystania z przeglądarki WWW

Następujący zestaw instrukcji umożliwia dostęp do narzędzia Xsockets z przeglądarki WWW. Aby utworzyć wiele instancji serwera, można zastosować te instrukcje w tym samym systemie wielokrotnie. Dzięki temu będzie jednocześnie uruchomionych wiele wersji, nasłuchujących na różnych portach. Aby skonfigurować Xsockets do korzystania z przeglądarki WWW, należy:

1. Skonfigurować serwer HTTP (oparty na Apache).
2. Skonfigurować serwer Tomcat.
3. Zaktualizować pliki konfiguracyjne.
4. Przetestować narzędzie Xsockets w przeglądarce.

Konfigurowanie serwera HTTP (opartego na Apache)

Aby skonfigurować przeglądarkę WWW do współpracy z narzędziem Xsockets, należy najpierw przeprowadzić konfigurację rodzimego narzędzia Xsockets. Poniżej zostało opisane konfigurowanie serwera HTTP (opartego na Apache), umożliwiające używanie narzędzia Xsockets za pośrednictwem przeglądarki WWW.

1. Sprawdź, czy instancja admin serwera HTTP pracuje w podsystemie QHTTPSVR. Jeśli nie, można ją uruchomić komendą CL
   ```
   STRTCPSVR SERVER(*HTTP) HTTPSVR(*ADMIN)
   ```
2. W przeglądarce WWW wpisz:
   ```
   http://<nazwa_systemu>:2001/
   ```
   gdzie <nazwa_systemu> to nazwa serwera iSeries. Na przykład: http://myiSeries:2001/.
3. Na stronie Zadania iSeries wybierz **IBM HTTP Server foriSeries**.
4. Z górnego menu wybierz zakładkę **Konfiguracja**.
5. Kliknij **Utwórz nowy serwer HTTP**.
6. Wybierz **Serwer HTTP (oparty na Apache)** i kliknij **Dalej**.
7. Wpisz nazwą instancji serwera. Ponieważ instancja będzie wyświetlała w przeglądarce strony narzędzia Xsocket, możesz użyć nazwy xsocket. Kliknij **Dalej**.
8. Wybierz **Nie**. Spowoduje to utworzenie nowej instancji serwera nie opartej na istniejącym serwerze. Kliknij **Dalej**.
9. Kliknij **Dalej**, aby zaakceptować domyślny katalog główny serwera.
10. Kliknij **Dalej**, aby zaakceptować domyślny katalog główny dokumentu.
11. Wybierz adres IP i dostępny port, którego chcesz użyć. Użyj portu o numerze większym od 1024. Kliknij **Dalej**.

    Uwaga: Nie należy wybierać domyślnego portu o numerze 80.
12. Wybierz **tak** lub **nie**, aby określić, czy chcesz dla tego serwera utworzyć protokół dostępu. Kliknij **Dalej**.
13. Na następnej stronie zostanie wyświetlona konfiguracja serwera HTTP (opartego na Apache). Jeśli te ustawienia są poprawne, kliknij **Zakończ**.
14. Kliknij **Zarządzaj nowo utworzonym serwerem**. Konfigurowanie serwera Apache dobiegło końca.

Następnie można:

Skonfigurować serwer Tomcat.

Konfigurowanie serwera Tomcat

Po skonfigurowaniu instancji serwera HTTP (opartego na Apache) należy skonfigurować serwer Tomcat, aby uruchomić narzędzie Xsockets w przeglądarce WWW.

1. W nagłówku **Dynamic content** wybierz opcję **ASF Tomcat Setup Server Task** .
2. Wybierz **Enable servlets for this HTTP Server**. Spowoduje to wpisanie danych do pliku definicji procesów roboczych. Kliknij **Next**.
3. Na stronie **Workers Definition** zaakceptuj wartości domyślne i kliknij **Next**.
4. Na stronie **URL to Worker Mapping** kliknij **Add**.
5. W kolumnie **URL(Mount Point)** wpisz /xsock. Kliknij **Continue**.
6. Kliknij **Add**.
7. W kolumnie **URL(Mount Point)** wpisz /xsock/*. Kliknij **Continue**.
8. Kliknij **Dalej**.
9. Na stronie **In-Process Application Context Definition** kliknij **Add**.
10. W kolumnie **URL Path** wpisz /xsock.
11. W kolumnie **Application Base Directory** wpisz webapps/xsock.
12. Kliknij **Continue**. Wyświetlony zostanie komunikat informujący o tym, że nie trzeba potrzeba podawać dalszych informacji.
13. Na stronie **Configure Application** kliknij **Configure**.
14. W nowo otwartym oknie przeglądarki w polu **Session Object timeout** wybierz 3 dni.

    Uwaga: Jest to wartość zalecana; parametr **Session Object timeout** może jednak przyjmować dowolne wartości.
15. Kliknij **Add**, aby dodać definicję serwletu i wykonać następujące czynności:
    a. W polu **Servlet class name** wpisz com.ibm.iseries.xsocket.XSocketServlet.
    b. W polu **URL patterns** wpisz /*.
    c. Ustaw wartość **Startup load sequence** na 3.
    d. Kliknij **Continue**.
    e. Kliknij **OK**. Spowoduje to zamknięcie okna przeglądarki.
16. W głównym oknie konfigurowania serwera Tomcat kliknij **Next**.
17. Kliknij **Finish**.
18. Kliknij **OK**. Konfigurowanie serwera Tomcat do obsługi narzędzia Xsockets dobiegło końca.

Następnie można:

Zaktualizować pliki konfiguracyjne.

Aktualizowanie plików konfiguracyjnych

Po skonfigurowaniu serwera Tomcat na serwerze HTTP (opartym na Apache) do obsługi narzędzia Xsockets należy ręcznie dokonać zmian w kilku plikach konfiguracyjnych dla instancji. Należy zaktualizować trzy pliki: web.xml, JAR i httpd.conf. Do wykonania tych czynności będą potrzebne następujące informacje:

- Nazwa biblioteki zawierającej pliki aplikacji Xsockets. Zostały one utworzone podczas wstępnego konfigurowania narzędzia Xsockets jako klienta rodzimego.
- Nazwa serwera utworzonego podczas konfigurowania serwera HTTP (opartego na Apache).

1. **Zaktualizuj plik web.xml**
    a. W wierszu komend wpisz
    ```
    wrklnk '/www/<nazwa_serwera>/webapps/xsock/WEB-INF/web.xml'
    ```
    gdzie <nazwa_serwera> jest nazwą instancji serwera utworzonej podczas konfigurowania serwera Apache. Na przykład, jeśli nazwą serwera jest xsocks, wpisz:
    ```
    wrklnk '/www/xsocks/webapps/xsock/WEB-INF/web.xml'
    ```

b. Aby przeprowadzić edycję pliku, naciśnij 2.
c. W pliku web.xml znajdź wiersz </servlet-class>.
d. Pod tym wierszem wpisz następujący kod:

```
<init-param>
          <param-name>library</param-name>
          <param-value>XXXX</param-value>
</init-param>
```

W miejscu znaków XXXX wpisz nazwę biblioteki utworzonej podczas konfigurowania narzędzia Xsockets.
e. Zapisz plik i zakończ sesję edycji.

2. **Przenieś plik JAR**
   a. W wierszu komend wpisz komendę:

```
CPY OBJ('/QSYS.LIB/XXXX.LIB/QATTIFS2.FILE/TSOXSOCK.MBR')
   TOOBJ('/www/<nazwa_serwera>/webapps/xsock/WEB-INF/lib/tsoxsock.jar')
   FROMCCSID(*OBJ) TOCCSID(819) OWNER(*NEW)
```

gdzie XXXX jest nazwą biblioteki utworzoną podczas konfigurowania narzędzia Xsockets, a <nazwa_serwera> jest nazwą instancji serwera utworzoną podczas konfigurowania serwera HTTP (opartego na Apache).

3. **Do pliku httpd.conf dodaj sprawdzanie uprawnień** (Ta czynność jest opcjonalna).
   Wymusi to uwierzytelnianie użytkowników próbujących skorzystać z aplikacji Xsockets przez serwer Apache.

   Uwaga: Niezbędne są także uprawnienia do zapisu podczas tworzenia gniazd UNIX.
   a. W wierszu komend wpisz

```
wrklnk '/www/<nazwa_serwera>/conf/httpd.conf'
```

gdzie <nazwa_serwera> jest nazwą instancji serwera utworzonej podczas konfigurowania serwera Apache. Na przykład, jeśli nazwą serwera jest xsocks, wpisz:

```
wrklnk '/www/xsocks/conf/httpd.conf''
```

   b. Aby przeprowadzić edycję pliku, naciśnij 2.
   c. Poniższe wiersze wstaw na końcu pliku.

```
<Location /xsock>
   AuthName "X Socket"
   AuthType Basic
   PasswdFile %%SYSTEM%%
   UserId %%CLIENT%%
   Require valid-user
   order allow,deny
   allow from all
</Location>
```

   d. Zapisz plik i zakończ sesję edycji.

Następnie można:

Przetestować narzędzie Xsockets w przeglądarce WWW.

Testowanie narzędzia Xsockets w przeglądarce WWW

Po zakończeniu ręcznej aktualizacji plików konfiguracyjnych można przetestować narzędzie Xsocket w przeglądarce.

1. Aby uruchomić instancję serwera, w wierszu komend wpisz:

```
STRTCPSVR SERVER(*HTTP) HTTPSVR(<nazwa_serwera>)
```

gdzie <nazwa_serwera> jest nazwą instancji serwera utworzonej podczas konfigurowania serwera Apache.

Uwaga: Może to zająć trochę czasu.

2. Sprawdź status, uruchamiając komendę WRKACTJOB z interfejsu wiersza komend. Jeśli wszystkie zadania z server_name mają status SIGW, można przejść do następnego kroku.
3. W przeglądarce WWW wpisz następujący adres URL:

```
http://<nazwa_systemu>:<port>/xsock/index
```

   gdzie <nazwa_serwera> i <port> to nazwa instancji serwera i numer portu określone podczas konfigurowania serwera Apache.
4. Po wyświetleniu zachęty wpisz nazwę użytkownika i hasło do serwera. Powinien zostać wyświetlony klient WWW narzędzia Xsocket.

Używanie Xsockets

Obecnie można pracować z narzędziem Xsockets na dwa sposoby. Można użyć rodzimego klienta lub przeglądarki WWW. Aby pracować z rodzimą wersją narzędzia Xsocket, należy je skonfigurować. Aby pracować z tym narzędziem w środowisku przeglądarki WWW, należy oprócz skonfigurowania samego narzędzia, wykonać czynności opisane w sekcji Konfigurowanie Xsockets do korzystania z przeglądarki WWW. Wiele pojęć jest podobnych w obu wersjach narzędzia. Oba narzędzia umożliwiają interaktywne wywoływanie funkcji gniazd i dostarczają informacji o numerach błędów zwracanych przez te funkcje. Jednakże ich interfejsy w pewnym stopniu różnią się. Poniższe instrukcje informują, jak należy pracować z narzędziem Xsocket w obu środowiskach.

Uwaga: Aby pracować z programami gniazd korzystającymi z funkcji API GSKit gniazd chronionych, konieczne jest używanie wersji WWW tego narzędzia.

Poniższe sekcje opisują, jak:

- Użyć rodzimego programu Xsockets
- Użyć programu Xsockets opartego na przeglądarce

Używanie rodzimego Xsockets

Przed wykonaniem tych czynności konieczne jest wykonanie wszystkich czynności konfiguracyjnych. Aby używać rodzimego klienta Xsocket:

1. W wierszu komend dodaj do listy bibliotek bibliotekę, w której znajduje się narzędzie Xsocket, wpisując komendę

```
ADDLIBLE <nazwa-biblioteki>
```

   gdzie <nazwa-biblioteki> jest nazwą biblioteki utworzonej podczas konfigurowania rodzimego klienta Xsockets. Na przykład, jeśli nazwą biblioteki jest MYXSOCKET, wpisz:

```
ADDLIBLE MYXSOCKET
```

2. W wierszu komend wpisz

```
CALL TSOXSOCK
```

3. Zostanie wyświetlone okno Xsockets, które umożliwi dostęp do wszystkich procedur gniazd poprzez pasek menu i pola wyboru. Okno to jest zawsze wyświetlane po wybraniu funkcji gniazd. Interfejsu tego można używać do wybierania istniejących programów używających gniazd. Aby pracować z nowym gniazdem, wykonaj następujące czynności:
   a. Z listy funkcji gniazda wybierz **socket** i naciśnij klawisz **Enter**.
   b. W wyświetlonym oknie **socket() prompt** wybierz odpowiednią rodzinę adresów, typ gniazda i protokół, a następnie naciśnij klawisz **Enter**.
   c. Wybierz **Descriptor**, a następnie **Select descriptor**.

      Uwaga: Jeśli istnieją już inne deskryptory gniazd, spowoduje to wyświetlenie listy deskryptorów aktywnych gniazd.
   d. Z wyświetlonej listy wybierz utworzony deskryptor gniazda.

      Uwaga: Jeśli istnieją inne deskryptory gniazd, narzędzie automatycznie zastosuje tę funkcję do najnowszego deskryptora gniazda.

4. Z listy funkcji gniazda wybierz tę, z którą chcesz pracować. Funkcja zostanie uruchomiona dla deskryptora wybranego w Kroku 3c. Po wybraniu funkcji gniazd zostanie wyświetlona grupa okien, w których można wprowadzić informacje właściwe dla danej funkcji gniazd. Jeśli na przykład zostanie wybrana funkcja **connect()**, w wyświetlonych oknach trzeba będzie podać długość adresu, rodzinę adresów i dane adresu. Wybrana funkcja gniazd jest następnie wywoływana z podanymi informacjami. Wszelkie błędy, jakie wystąpią przy wywołaniu funkcji zostaną wyświetlone jako kody błędów errno.

Uwagi:

1. Narzędzie Xsockets używa obsługi adapteru DDS. Dlatego sposób wprowadzania danych i realizacji wyborów w oknach lub na ekranach zależy od tego, czy używasz terminalu graficznego, czy tekstowego. Na przykład, na terminalu graficznym będzie widoczne pole wyboru funkcji gniazda; w przeciwnym razie będzie to zwykłe pole.
2. Należy wiedzieć, że w gnieździe są dostępne żądania **ioctl()**, które nie zostały zaimplementowane w narzędziu.

Używanie Xsockets w przeglądarce WWW

Aby pracować z narzędziem Xsockets w przeglądarce WWW, należy przeprowadzić konfigurowanie rodzimej wersji Xsockets i wszystkie czynności związane z konfigurowaniem przeglądarki WWW. Aby pracować z narzędziem Xsockets z poziomu przeglądarki WWW, wykonaj następujące czynności:

1. W przeglądarce WWW wpisz:
   ```
   http://nazwa-serwera:2001/
   ```
   gdzie nazwa-serwera to nazwa systemu iSeries, na którym pracuje instancja serwera.
2. Wybierz **Administration**.
3. Z lewego paska nawigacyjnego wybierz opcję **Manage HTTP Servers**.
4. Wybierz nazwę instancji i kliknij **Start**. Możesz także uruchomić instancję serwera z wiersza komend, wpisując:
   ```
   STRTCPSVR SERVER(*HTTP) HTTPSVR(<nazwa_instancji>)
   ```
   gdzie <nazwa_instancji> jest nazwą serwera HTTP utworzonego podczas konfigurowania serwera Apache. Na przykład można użyć instancji serwera o nazwie xsocks.
5. Aby wyświetlić aplikację Xsockets, w przeglądarce wpisz poniższy adres URL:
   ```
   http://<nazwa_systemu>:<port>/xsock/index
   ```
   gdzie <nazwa_systemu> to nazwa serwera iSeries, a <port> to port podany podczas tworzenia instancji serwera HTTP. Na przykład, jeśli nazwą systemu jest myiSeries, a instancja serwera HTTP nasłuchuje na porcie 1025, wpisz:
   ```
   http://myiSeries:1025/xsock/index
   ```
6. Po załadowaniu narzędzia Xsocket w przeglądarce można pracować z istniejącym deskryptorem gniazda lub utworzyć nowe. Wiele pojęć jest podobnych w obu wersjach narzędzia. Oba narzędzia umożliwiają interaktywne wywoływanie funkcji gniazd i dostarczają informacji o numerach błędów zwracanych przez te funkcje. Jednakże ich interfejsy w pewnym stopniu różnią się. Aby utworzyć nowy deskryptor gniazda:
   a. W menu **Xsocket** wybierz **gniazdo**.
   b. W wyświetlonym oknie **Xsocket Query** wybierz odpowiednią rodzinę adresów, typ gniazda i protokół dla tego deskryptora gniazda. Kliknij **Submit**.
   c. Po ponownym załadowaniu strony nowy deskryptor gniazda zostanie wyświetlony w menu rozwijanym **Socket**.
   d. W menu **Xsocket** wybierz wywołania funkcji, które mają być stosowane dla tego deskryptora gniazda. Podobnie jak rodzima wersja Xsockets, jeśli nie wybierzesz deskryptora gniazda, narzędzie zastosuje wywołania funkcji do najnowszego deskryptora gniazda.

Usuwanie obiektów utworzonych przez narzędzie Xsocket

Może zajść potrzeba usunięcia obiektów utworzonych przez narzędzie Xsockets. Program instalacyjny tworzy program o nazwie TSODLT, który służy do usuwania obiektów utworzonych przez Xsockets (z wyjątkiem biblioteki i samego programu TSODLT) i/lub usuwania podzbiorów źródłowych utworzonych przez Xsockets. Opisany poniżej zestaw komend umożliwia usuwanie obiektów:

Aby usunąć TYLKO podzbiory źródłowe używane przez narzędzie, wpisz:

```
CALL TSODLT   (*YES *NONE)
```

Aby usunąć TYLKO obiekty utworzone przez narzędzie, wpisz:

```
CALL TSODLT   (*NO  nazwa-biblioteki)
```

Aby usunąć ZARÓWNO podzbiory źródłowe, jaki i obiekty utworzone przez narzędzie, wpisz:

```
CALL TSODLT   (*YES nazwa-biblioteki)
```

Dostosowywanie Xsockets

Narzędzie Xsockets można zmienić dodając obsługę dla sieciowych procedur gniazd, na przykład dla **inet_addr()**. W przypadku dostosowywania narzędzia do własnych potrzeb nie zaleca się dokonywania zmian w bibliotece QUSRTOOL. Należy skopiować zbiory źródłowe do osobnej biblioteki i tam dokonać zmian. Dzięki temu w bibliotece QUSRTOOL zostaną zachowane oryginalne zbiory na wypadek, gdyby były potrzebne w przyszłości. Do ponownej kompilacji narzędzia po dokonaniu zmian można użyć programu TSOCRT (jeśli zbiory źródłowe zostały skopiowane do osobnej biblioteki, należy również dokonać zmian w programie TSOCRT). Do usunięcia starych wersji obiektów narzędzia przed utworzeniem nowej wersji służy program TSODLT.

Narzędzia do serwisowania

W związku z ciągłym wzrostem zastosowań gniazd i gniazd chronionych w aplikacjach i serwerach używanych na potrzeby e-biznesu, rosną również potrzeby związane z rozwojem narzędzi serwisowych. Udoskonalone narzędzia serwisowe umożliwiają śledzenie działania programów używających gniazd w celu znalezienia rozwiązań problemów związanych z aplikacjami używającymi gniazd i warstwy SSL. Narzędzia te umożliwiają programistom i pracownikom centrum obsługi umiejscowienie problemów z użyciem gniazd poprzez wybór takich parametrów gniazd, jak adres IP lub informacje o porcie.

Poniższa tabela stanowi przegląd informacji o tych narzędziach serwisowych.

Tabela 21. Narzędzia serwisowe dla gniazd i gniazd chronionych

Narzędzie serwisowe	Opis
Filtrowanie śledzenia LIC (TRCINT i TRCCNN)	Udostępnia selektywne śledzenie gniazda. Obecnie możliwe jest ograniczenie śledzenia do rodziny adresów, typu gniazda, adresu IP i informacji o porcie. Można również ograniczyć śledzenie do pewnych kategorii funkcji API gniazd, a także do tylko tych gniazd, które mają ustawioną opcję SO_DEBUG. Od wersji V5R2 śledzenie LIC można filtrować pod kątem wątku, zadania, profilu użytkownika, nazwy zadania i nazwy serwera.
Śledzenie zadania za pomocą komendy STRTRC SSNID(*GEN) JOBTRCTYPE(*TRCTYPE) TRCTYPE((*SOCKETS *ERROR))	Od wersji V5R2 komenda STRTRC udostępnia dodatkowe parametry, dzięki którym dane wyjściowe są oddzielone od innych, niezwiązanych z gniazdami punktów śledzenia. Dane te zawierają kod zakończenia i kod błędu w przypadku wystąpienia błędu podczas operacji gniazda. Szczegóły zawiera sekcja Opis komendy STRTRC (Start Trace - Uruchomienie śledzenia) w Centrum informacyjnym.
Śledzenie rejestratora przebiegu przetwarzania	Informacje ze śledzenia komponentu LIC gniazd będą teraz obejmowały zrzut pozycji rejestratora przebiegu przetwarzania dla każdej wykonanej operacji gniazda.
Powiązane informacje o zadaniu	Umożliwiają pracownikom serwisowym i programistom znalezienie wszystkich zadań powiązanych z gniazdem połączonym lub nasłuchującym. Dla aplikacji używających gniazd z rodziny adresów AF_INET lub AF_INET6 informacje te mogą być wyświetlone za pomocą komendy NETSTAT.

Tabela 21. Narzędzia serwisowe dla gniazd i gniazd chronionych (kontynuacja)

Status połączenia (Opcja 3 komendy NETSTAT) udostępnia parametr SO_DEBUG	Udostępnia rozszerzone informacje debugowania niskiego poziomu, jeśli dla aplikacji używającej gniazd zostanie ustawiona opcja SO_DEBUG.
Kod zakończenia i przetwarzanie komunikatów gniazd chronionych	Wyświetla standardowe komunikaty kodów powrotu gniazd chronionych przy użyciu dwóch funkcji API SSL_. Funkcje te to **SSL_Strerror()** i **SSL_Perror()**. Ponadto funkcja **gsk_strerror()** udostępnia podobne możliwości co funkcje API GSKit. Dostępna jest także funkcja API **hstrerror()**, dostarczająca informacji o kodach powrotu procedur tłumaczenia nazw.
Punkty śledzenia Performance Data Collection (PDC)	Udostępnia informacje ze śledzenia przepływu danych z aplikacji poprzez gniazda i stos TCP/IP.

Informacje pokrewne

Poniżej znajduje się lista dokumentacji technicznej (Redbooks) IBM (w formacie PDF), serwisów WWW i tematów w Centrum informacyjnym, które zawierają więcej informacji o programowaniu z użyciem gniazd. Dokumenty te można wyświetlać na ekranie i drukować.

Dokumentacja techniczna (Redbooks) IBM

- Who Knew You Could Do That with RPG IV? A Sorcerer's Guide to System Access and More
- IBM @server iSeries Wired Network Security: OS/400 V5R1 DCM and Cryptographic Enhancements

Dokumenty RFC (Request For Comments)

- **IPv6**
  - RFC 2553: ″Basic Socket Interface Extensions for IPv6″
  - RFC 2292: ″Advanced Sockets API for IPv6″
- **DNS (Domain Name System)**
  - 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)″
- **Warstwa SSL/ochrona warstwy transportowej**
  - RFC 2246: ″The TLS Protocol Version 1.0 ″

Inne zasoby sieci WWW

- Technical Standard: Networking Services (XNS), Issue 5.2 Draft 2.0.

Informacje dotyczące kodu

Niniejszy dokument zawiera przykładowe kody programów.

IBM udziela Użytkownikowi niewyłącznej licencji w zakresie praw autorskich na używanie wszystkich przykładów kodu programistycznego, z których może on generować podobne funkcje, dostosowane do własnych, specyficznych potrzeb.

Cały kod przykładowy jest dostarczany przez IBM wyłącznie do celów obrazowania. Programy przykładowe nie zostały gruntownie przetestowane. IBM nie może zatem gwarantować lub sugerować niezawodności, użyteczności i funkcjonalności tych programów.

Wszelkie zawarte tutaj programy są dostarczane w stanie, w jakim się znajdują (″AS IS″) bez udzielania jakichkolwiek gwarancji. Nie udziela się domniemanych gwarancji nienaruszania praw osób trzecich, gwarancji przydatności handlowej ani też przydatności do określonego celu.

Uwagi

Niniejsza publikacja została przygotowana z myślą o produktach i usługach oferowanych w Stanach Zjednoczonych.

IBM może nie oferować w innych krajach produktów, usług lub opcji omawianych w tej publikacji. Informacje o produktach i usługach dostępnych w danym kraju można uzyskać od lokalnego przedstawiciela IBM. Odwołanie do produktu, programu lub usługi IBM nie oznacza, że można użyć wyłącznie tego produktu, programu lub usługi. Zamiast nich można zastosować ich odpowiednik funkcjonalny pod warunkiem, że nie narusza to praw własności intelektualnej IBM. Jednakże cała odpowiedzialność za ocenę przydatności i sprawdzenie działania produktu, programu lub usługi pochodzących od producenta innego niż IBM spoczywa na użytkowniku.

IBM może posiadać patenty lub złożone wnioski patentowe na towary i usługi, o których mowa w niniejszej publikacji. Przedstawienie niniejszej publikacji nie daje żadnych uprawnień licencyjnych do tychże patentów. Pisemne zapytania w sprawie licencji można przesyłać na adres:

IBM Director of Licensing
IBM Corporation
500 Columbus Avenue
Thornwood, NY 10594-1785
U.S.A.

Zapytania w sprawie licencji na informacje dotyczące zestawów znaków dwubajtowych (DBCS) należy kierować do lokalnych działów własności intelektualnej IBM (IBM Intellectual Property Department) lub zgłaszać na piśmie pod adresem:

IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106, Japan

Poniższy akapit nie obowiązuje w Wielkiej Brytanii, a także w innych krajach, w których jego treść pozostaje w sprzeczności z przepisami prawa miejscowego: INTERNATIONAL BUSINESS MACHINES CORPORATION DOSTARCZA TĘ PUBLIKACJĘ W TAKIM STANIE, W JAKIM SIĘ ZNAJDUJE (“ AS IS”) BEZ UDZIELANIA JAKICHKOLWIEK GWARANCJI (W TYM TAKŻE RĘKOJMI), WYRAŹNYCH LUB DOMNIEMANYCH, A W SZCZEGÓLNOŚCI DOMNIEMANYCH GWARANCJI PRZYDATNOŚCI HANDLOWEJ, PRZYDATNOŚCI DO OKREŚLONEGO CELU ORAZ GWARANCJI, ŻE PUBLIKACJA NIE NARUSZA PRAW OSÓB TRZECICH. Ustawodawstwa niektórych krajów nie dopuszczają zastrzeżeń dotyczących gwarancji wyraźnych lub domniemanych w odniesieniu do pewnych transakcji; w takiej sytuacji powyższe zdanie nie ma zastosowania.

Informacje zawarte w niniejszej publikacji mogą zawierać nieścisłości techniczne lub błędy drukarskie. Informacje te są okresowo aktualizowane, a zmiany te zostaną ujęte w kolejnych wydaniach tej publikacji. IBM zastrzega sobie prawo do wprowadzania ulepszeń i/lub zmian w produktach i/lub programach opisanych w tej publikacji w dowolnym czasie, bez wcześniejszego powiadomienia.

Wszelkie wzmianki w tej publikacji na temat stron internetowych innych firm zostały wprowadzone wyłącznie dla wygody użytkowników i w żadnym wypadku nie stanowią zachęty do ich odwiedzania. Materiały dostępne na tych stronach nie są częścią materiałów opracowanych dla tego produktu IBM, a użytkownik korzysta z nich na własną odpowiedzialność.

IBM ma prawo do używania i rozpowszechniania informacji przysłanych przez użytkownika w dowolny sposób, jaki uzna za właściwy, bez żadnych zobowiązań wobec ich autora.

© Copyright IBM Corp. 2001, 2005

Licencjobiorcy tego programu, którzy chcieliby uzyskać informacje na temat programu w celu: (i) wdrożenia wymiany informacji między niezależnie utworzonymi programami i innymi programami (łącznie z tym opisywanym) oraz (ii) wspólnego wykorzystywania wymienianych informacji, powinni skontaktować się z:

IBM Corporation
Software Interoperability Coordinator, Department 49XA
3605 Highway 52 N
Rochester, MN 55901
U.S.A.

Informacje takie mogą być udostępnione, o ile spełnione zostaną odpowiednie warunki, w tym, w niektórych przypadkach, uiszczenie odpowiedniej opłaty.

Licencjonowany program opisany w niniejszej publikacji oraz wszystkie inne licencjonowane materiały dostępne dla tego programu są dostarczane przez IBM na warunkach określonych w Umowie IBM z Klientem, Międzynarodowej Umowie Licencyjnej IBM na Program lub w innych podobnych umowach zawartych między IBM i użytkownikami.

Wszelkie dane dotyczące wydajności zostały zebrane w kontrolowanym środowisku. W związku z tym rezultaty uzyskane w innych środowiskach operacyjnych mogą się znacząco różnić. Niektóre pomiary mogły być dokonywane na systemach będących w fazie rozwoju i nie ma gwarancji, że pomiary te wykonane na ogólnie dostępnych systemach dadzą takie same wyniki. Niektóre z pomiarów mogły być estymowane przez ekstrapolację. Rzeczywiste wyniki mogą być inne. Użytkownicy powinni we własnym zakresie sprawdzić odpowiednie dane dla ich środowiska.

Informacje dotyczące produktów firm innych niż IBM pochodzą od dostawców tych produktów, z opublikowanych przez nich zapowiedzi lub innych powszechnie dostępnych źródeł. Firma IBM nie testowała tych produktów i nie może potwierdzić dokładności pomiarów wydajności, kompatybilności ani żadnych innych danych związanych z tymi produktami. Pytania dotyczące produktów firm innych niż IBM należy kierować do dostawców tych produktów.

Wszelkie stwierdzenia dotyczące przyszłych kierunków rozwoju i zamierzeń IBM mogą zostać zmienione lub wycofane bez powiadomienia.

Publikacja ta zawiera przykładowe dane i raporty używane w codziennych operacjach działalności gospodarczej. W celu kompleksowego ich zilustrowania, podane przykłady zawierają nazwiska osób prywatnych, nazwy przedsiębiorstw oraz nazwy produktów. Wszystkie te nazwy są fikcyjne i jakiekolwiek ich podobieństwo do nazwisk, nazw i adresów używanych w rzeczywistych przedsiębiorstwach jest całkowicie przypadkowe.

LICENCJA NA PRAWA AUTORSKIE:

Niniejsza publikacja zawiera przykładowe aplikacje w kodzie źródłowym, ilustrujące techniki programowania w różnych systemach operacyjnych. Użytkownik może kopiować, modyfikować i dystrybuować te programy przykładowe w dowolnej formie bez uiszczania opłat na rzecz IBM, w celu projektowania, używania, sprzedaży lub dystrybucji aplikacji zgodnych z aplikacyjnym interfejsem programowym dla tego systemu operacyjnego, dla którego napisane zostały programy przykładowe. Programy przykładowe nie zostały gruntownie przetestowane. IBM nie może zatem gwarantować lub sugerować niezawodności, użyteczności i funkcjonalności tych programów. Użytkownik może kopiować, modyfikować i dystrybuować te programy przykładowe w dowolnej formie bez uiszczania opłat na rzecz IBM, w celu projektowania, używania, sprzedaży lub dystrybucji aplikacji zgodnych z aplikacyjnym interfejsem programowym IBM.

Każda kopia programu przykładowego lub jakikolwiek jego fragment, jak też jakiekolwiek prace pochodne muszą zawierać następujące uwagi dotyczące praw autorskich:

© (IBM) (2004). Fragmenty tego kodu pochodzą z Programów przykładowych IBM Corp. © Copyright IBM Corp. 2001, 2004. Wszelkie prawa zastrzeżone.

Znaki towarowe

Następujące nazwy są znakami towarowymi International Business Machines Corporation w Stanach Zjednoczonych i/lub w innych krajach:

Anynet
C/400
IBM
iSeries
Language Environment
Operating System/400
OS/400
Redbooks

Microsoft, Windows, Windows NT i logo Windows są znakami towarowymi Microsoft Corporation w Stanach Zjednoczonych i/lub w innych krajach.

Java oraz wszystkie znaki towarowe dotyczące języka Java są znakami towarowymi Sun Microsystems, Inc. w Stanach Zjednoczonych i/lub w innych krajach.

UNIX jest zastrzeżonym znakiem towarowym The Open Group w Stanach Zjednoczonych i/lub w innych krajach.

Nazwy innych firm, produktów i usług mogą być znakami towarowymi lub znakami usług innych podmiotów.

Warunki pobierania i drukowania publikacji

Zezwolenie na korzystanie z publikacji, które Użytkownik zamierza pobrać, jest przyznawane na poniższych warunkach. Warunki te wymagają akceptacji Użytkownika.

Użytek osobisty: Użytkownik ma prawo kopiować te publikacje do własnego, niekomercyjnego użytku pod warunkiem zachowania wszelkich uwag dotyczących praw własności. Użytkownik nie ma prawa dystrybuować ani wyświetlać tych publikacji czy ich części, ani też wykonywać z nich prac pochodnych bez wyraźnej zgody IBM.

Użytek służbowy: Użytkownik ma prawo kopiować te publikacje, dystrybuować je i wyświetlać wyłącznie w ramach przedsiębiorstwa Użytkownika pod warunkiem zachowania wszelkich uwag dotyczących praw własności. Użytkownik nie ma prawa wykonywać z tych publikacji ani z ich części prac pochodnych, kopiować ich, dystrybuować ani wyświetlać poza przedsiębiorstwem Użytkownika bez wyraźnej zgody IBM.

Z wyjątkiem zezwoleń wyraźnie udzielonych w niniejszym dokumencie, nie udziela się jakichkolwiek innych zezwoleń, licencji ani praw, wyraźnych czy domniemanych, odnoszących się do tych publikacji czy jakichkolwiek informacji, danych, oprogramowania lub innej własności intelektualnej, o których mowa w niniejszym dokumencie.

IBM zastrzega sobie prawo wycofania udzielonych niniejszym zezwoleń w dowolnym momencie, w którym, wedle uznania IBM, używanie Publikacji nie służy interesom IBM lub z ustaleń IBM wynika, że powyższe instrukcje nie są przestrzegane.

Użytkownik ma prawo pobierać, eksportować lub reeksportować niniejsze informacje pod warunkiem zachowania bezwzględnej i pełnej zgodności z obowiązującym prawem i przepisami, w tym ze wszelkimi prawami i przepisami eksportowymi Stanów Zjednoczonych. IBM NIE UDZIELA ŻADNYCH GWARANCJI ODNOŚNIE ZAWARTOŚCI TYCH PUBLIKACJI. PUBLIKACJE TE SĄ DOSTARCZANE W STANIE, W JAKIM SIĘ ZNAJDUJĄ (″AS-IS″) BEZ UDZIELANIA JAKICHKOLWIEK GWARANCJI, W TYM TAKŻE RĘKOJMI, WYRAŹNYCH CZY DOMNIEMANYCH, A W SZCZEGÓLNOŚCI DOMNIEMANYCH GWARANCJI PRZYDATNOŚCI HANDLOWEJ CZY PRZYDATNOŚCI DO OKREŚLONEGO CELU.

Wszystkie materiały są chronione prawem autorskim IBM Corporation.

Pobieranie lub drukowanie publikacji z tego serwisu oznacza zgodę na warunki zawarte w niniejszym dokumencie.

IBM