System i

Programowanie
Programowanie z użyciem gniazd

Wersja 6 wydanie 1

System i

Programowanie
Programowanie z użyciem gniazd

Wersja 6 wydanie 1

Uwaga

Przed skorzystaniem z tych informacji oraz z produktu, którego dotyczą, należy przeczytać informacje zawarte w sekcji “Uwagi”, na stronie 201.

To wydanie dotyczy systemu operacyjnego IBM i5/OS wersja 6, wydanie 1, modyfikacja 0 (nr produktu 5761–SS1) oraz wszystkich kolejnych wydań i modyfikacji, chyba że w nowych wydaniach zostanie określone inaczej. Wersja ta nie działa na wszystkich modelach komputerów z procesorem RISC ani na modelach z procesorem CISC.

Spis treści

Programowanie z użyciem gniazd

Gniazdo to punkt obsługi połączenia komunikacyjnego (punkt końcowy), który można nazwać i dowiązać do adresu w sieci. W sekcji Programowanie z użyciem gniazd przedstawiono wykorzystanie funkcji API gniazd do ustanawiania połączeń między procesami zdalnym i lokalnym.

Procesy korzystające z gniazd mogą rezydować w tym samym systemie bądź w różnych systemach w różnych sieciach. Gniazda są przydatne zarówno dla aplikacji samodzielnych, jak i sieciowych. Gniazda umożliwiają wymianę informacji między procesami na tym samym komputerze lub poprzez sieć, pozwalają na dystrybucję zadań do najbardziej wydajnego komputera oraz ułatwiają dostęp do danych przechowywanych na serwerze. Funkcje API gniazd są standardem sieciowym dla protokołu TCP/IP. Funkcje te są obsługiwane w wielu systemach operacyjnych. Gniazda systemu i5/OS obsługują wiele protokołów transportowych i sieciowych. Funkcje systemowe i funkcje sieciowe gniazd realizują ochronę wątków.

Programiści pracujący w zintegrowanym środowisku językowym (Integrated Language Environment - ILE) C mogą korzystać z tej kolekcji tematów przy opracowywaniu aplikacji używających gniazd. Programowanie z wykorzystaniem funkcji API gniazd jest także możliwe dla innych języków w środowisku ILE, na przykład dla RPG.

Interfejs programowania z użyciem gniazd jest także obsługiwany w języku Java.

Uwaga: Korzystając z przykładów, użytkownik wyraża zgodę na warunki podane w sekcji “Licencja na kod oraz Informacje dotyczące kodu” na stronie 199.

Pliki PDF z informacjami na temat programowania z użyciem gniazd

Informacje zawarte w tym temacie są także dostępne w postaci pliku PDF, który można wyświetlić i wydrukować.

Aby wyświetlić lub pobrać ten dokument jako plik PDF, kliknij odsyłacz Programowanie z użyciem gniazd (około 1975 kB).

Inne informacje

Dokumenty te można wyświetlać na ekranie i drukować.

Dokumentacja techniczna IBM (Redbooks):

- Who Knew You Could Do That with RPG IV? A Sorcerer’s Guide to System Access and More (5630 kB)
- IBM eServer iSeries Wired Network Security: OS/400 V5R1 DCM and Cryptographic Enhancements (10 035 kB)

Można przeglądać lub pobrać następujące tematy pokrewne:

- **IPv6**
  - RFC 3493: ″Basic Socket Interface Extensions for IPv6″
  - RFC 3513: ″Internet Protocol Version 6 (IPv6) Addressing Architecture″
  - RFC 3542: ″Advanced Sockets Application Program Interface (API) for IPv6″
- **System nazw domen**
  - 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

Zapisywanie plików PDF

Aby zapisać plik PDF na stacji roboczej w celu jego wyświetlenia lub wydrukowania, wykonaj następujące czynności:

1. Kliknij prawym przyciskiem myszy odsyłacz do pliku PDF w przeglądarce.
2. Kliknij opcję zapisania pliku PDF lokalnie.
3. Przejdź do katalogu, w którym ma zostać zapisany plik PDF.
4. Kliknij opcję **Zapisz**.

Pobieranie programu Adobe Reader

Do przeglądania i drukowania plików PDF potrzebny jest program Adobe Reader. Bezpłatną kopię tego programu można pobrać z serwisu WWW firmy Adobe (www.adobe.com/products/acrobat/readstep.html) .

Wymagania wstępne dla programowania z użyciem gniazd

Przed przystąpieniem do tworzenia aplikacji używających gniazd należy wykonać poniższe czynności, tak aby spełnione zostały wymagania dotyczące kompilatora, rodzin adresów AF_INET i AF_INET6 oraz funkcji API SSL i funkcji API z zestawu GSKit.

Wymagania dotyczące kompilatora

1. Zainstaluj bibliotekę QSYSINC. Biblioteka ta zawiera pliki nagłówkowe, niezbędne przy kompilowaniu aplikacji używających gniazd.
2. Zainstaluj program licencjonowany ILE C (5761-WDS, opcja 51).

Wymagania dotyczące rodzin adresów AF_INET i AF_INET6

Po spełnieniu wymagań dotyczących kompilatora należy wykonać następujące czynności:

1. Zaplanuj instalację TCP/IP.
2. Zainstaluj TCP/IP.
3. Skonfiguruj TCP/IP po raz pierwszy.
4. Skonfiguruj IPv6 dla protokołu TCP/IP, jeśli zamierzasz pisać aplikacje korzystające z rodziny adresów AF_INET6.

Wymagania dotyczące funkcji API SSL i funkcji API GSKit

Po spełnieniu wymagań związanych z kompilatorem oraz rodzinami adresów AF_INET i AF_INET6 wykonaj następujące czynności:

1. Zainstaluj i skonfiguruj program licencjonowany Digital Certificate Manager (5761–SS1, opcja 34). Więcej informacji na ten temat można znaleźć w sekcji Program Digital Certificate Manager w Centrum informacyjnym.
2. Aby używać SSL ze sprzętem szyfrującym, zainstaluj i skonfiguruj produkt 2058 Cryptographic Accelerator, 4758 Cryptographic Coprocessor lub 4764 Cryptographic Coprocessor. Produkt 2058 Cryptographic Accelerator umożliwia przeniesienie obciążenia związanego z szyfrowaniem SSL z systemu operacyjnego na kartę. Produkt 4758 Cryptographic Coprocessor może służyć do szyfrowania SSL, jednak w przeciwieństwie do produktu 2058 udostępnia on więcej funkcji szyfrujących, na przykład szyfrowanie i deszyfrowanie kluczy. Produkt 4764 Cryptographic Coprocessor jest lepszą wersją produktu 4758 Cryptographic Coprocessor. Więcej informacji na temat produktów 2058 Cryptographic Accelerator, 4758 Cryptographic Coprocessor i 4764 Cryptographic Coprocessor można znaleźć w sekcji Cryptography.

   Odsyłacze pokrewne

   “Korzystanie z rodziny adresów AF_INET” na stronie 27
   Gniazda korzystające z rodziny adresów AF_INET mogą być zorientowane na połączenie (typ SOCK_STREAM) lub bezpołączeniowe (typ SOCK_DGRAM). Gniazda AF_INET zorientowane na połączenie używają jako protokołu transportowego TCP. Bezpołączeniowe gniazda AF_INET używają jako protokołu transportowego UDP.

   “Korzystanie z rodziny adresów AF_INET6” na stronie 27
   Gniazda AF_INET6 zapewniają obsługę 128-bitowych (16-bajtowych) struktur adresów protokołu IP w wersji 6 (IPv6). Programiści mogą tworzyć aplikacje korzystające z rodziny adresów AF_INET6, które będą akceptowały połączenia z węzłów klienckich obsługujących zarówno protokół IPv4, jak i IPv6 lub tylko protokół IPv6.

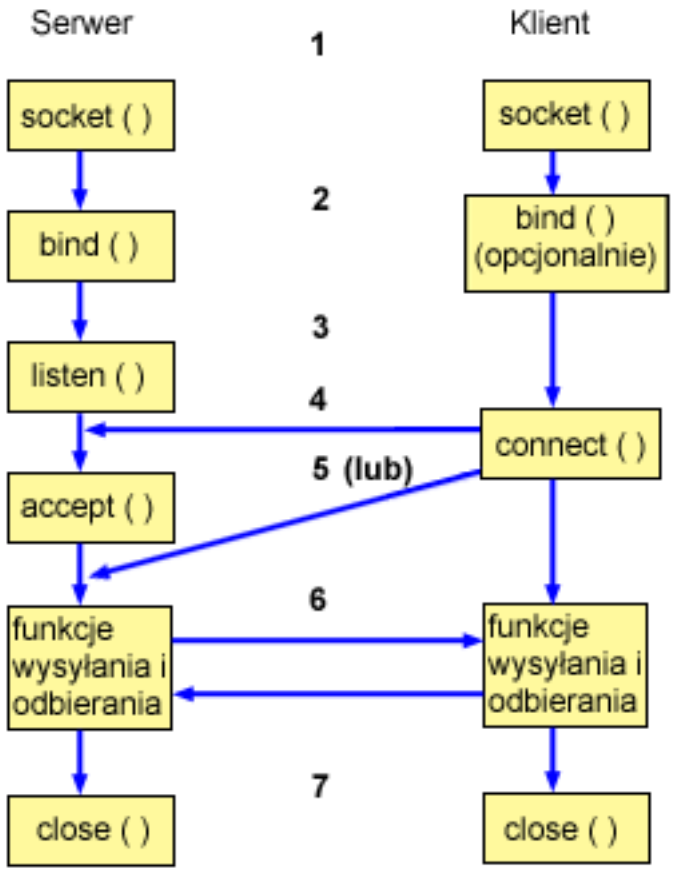
Działanie gniazd

Gniazd używa się powszechnie do obsługi interakcji między klientem a serwerem. W typowej konfiguracji systemu serwer jest umieszczony na jednym komputerze, a klienty na innych komputerach. Klienty łączą się z serwerem, wymieniają informacje, a następnie się odłączają.

Ustanawianie gniazd wiąże się z typowym przebiegiem zdarzeń. W modelu klient/serwer zorientowanym na połączenie gniazdo procesu serwera oczekuje na żądania od klienta. W tym celu serwer najpierw ustanawia (wiąże) adres, z którego może skorzystać klient w celu znalezienia serwera. Kiedy adres jest ustanowiony, serwer czeka, aż klient zażąda usługi. Wymiana danych między klientem a serwerem zachodzi wtedy, gdy klient łączy się z serwerem poprzez gniazdo. Serwer spełnia żądanie klienta i odsyła do niego odpowiedź.

Uwaga: IBM obsługuje obecnie dwie wersje większości funkcji API gniazd. Domyślne gniazda systemu i5/OS używają struktury i składni BSD (Berkeley Socket Distribution) 4.3. W drugiej wersji gniazd używa się składni i struktur zgodnych z BSD 4.4 i specyfikacją interfejsu programistycznego UNIX 98. Aby korzystać z interfejsu zgodnego ze standardem UNIX 98, można określić makro _XOPEN_SOURCE.

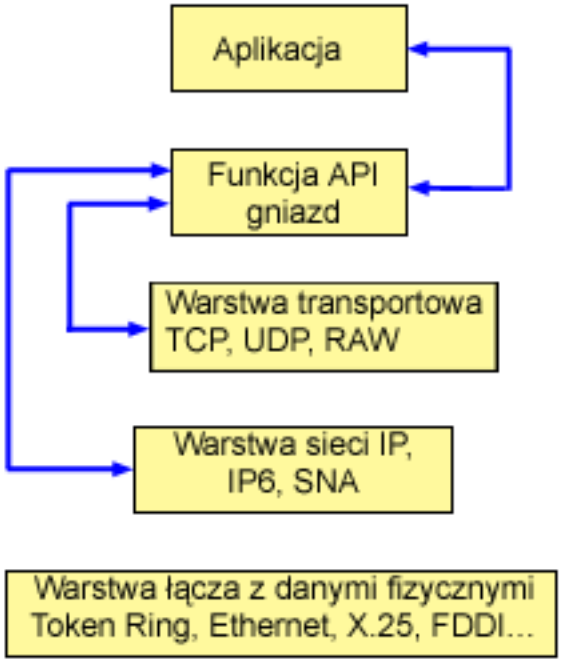
Na poniższym rysunku przedstawiono typowy przebieg zdarzeń (i sekwencję wywoływanych funkcji API) sesji tworzenia gniazda zorientowanego na połączenie. Wyjaśnienie każdego zdarzenia znajduje się w dalszej części sekcji.



Oto typowy przebieg zdarzeń dla gniazd zorientowanych na połączenie:

1. Funkcja API socket() tworzy punkt końcowy komunikacji i zwraca deskryptor gniazda reprezentujący ten punkt.
2. Kiedy aplikacja ma deskryptor gniazda, może powiązać z gniazdem unikalną nazwę. Serwery muszą powiązać nazwę, aby mogły być dostępne z sieci.
3. Funkcja API listen() wskazuje na gotowość do zaakceptowania żądań połączenia wysyłanych przez klienta. Kiedy dla gniazda zostanie wywołana funkcja API listen(), nie może ono aktywnie inicjować żądań połączenia. Funkcja API listen() jest wywoływana po przydzieleniu gniazda za pomocą funkcji API socket() i po powiązaniu nazwy z gniazdem za pomocą funkcji API bind(). Funkcję API listen() należy wywołać przed wywołaniem funkcji API accept().
4. Aplikacja klienta ustanawia połączenie z serwerem za pomocą funkcji connect() w gnieździe strumieniowym.
5. Aplikacja serwera odbiera żądanie połączenia z klienta za pomocą funkcji API accept(). Zanim serwer wywoła funkcję API accept(), musi pomyślnie wywołać funkcje API bind() i listen().
6. Po ustanowieniu połączenia między gniazdami strumieniowymi (na kliencie i na serwerze) można używać dowolnych funkcji API gniazd, służących do przesyłania danych. Istnieje wiele funkcji API służących do przesyłania danych przez klienty i serwery, na przykład send(), recv(), read(), write() i inne.
7. Kiedy klient lub serwer zamierza zakończyć działanie, wywołuje funkcję API close(), co powoduje zwolnienie zasobów systemu uzyskanych przez gniazdo.

Uwaga: Funkcje API gniazd w modelu komunikacyjnym znajdują się między warstwą aplikacji a warstwą transportową. Nie stanowią one warstwy w modelu komunikacyjnym. W typowym modelu komunikacyjnym funkcje API umożliwiają interakcję aplikacji z warstwą transportową i sieciową. Strzałki na poniższym rysunku pokazują pozycje gniazda i warstwę komunikacyjną obsługiwaną przez gniazdo.



Konfiguracja sieci nie pozwala zwykle na połączenia między chronioną siecią wewnętrzną a niechronioną siecią zewnętrzną. Można jednak pozwolić gniazdom na komunikowanie się z programami typu serwer działającymi w systemie poza firewallem (bardzo bezpiecznym hostem).

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

Odsyłacze pokrewne

"Zgodność z Berkeley Software Distribution (BSD)" na stronie 66
Gniazda są interfejsem systemu BSD.

"Zgodność ze standardem UNIX 98" na stronie 69
Standard UNIX 98, opracowany przez konsorcjum programistów i dostawców oprogramowania Open Group, usprawnił współdziałanie w ramach systemu operacyjnego UNIX. Uwzględniona w nim została większość funkcji związanych z obsługą Internetu, dzięki którym system UNIX stał się znany.

Informacje pokrewne

Funkcja API socket() - tworzenie gniazd

Funkcja API listen() - nasłuchiwanie przychodzących żądań połączenia

Funkcja API bind() - ustawianie lokalnego adresu gniazda

Funkcja API accept() - oczekiwanie na żądanie i nawiązywanie połączenia

Funkcja API send() - wysyłanie danych

Funkcja API recv() - odbieranie danych

Funkcja API close() - zamykanie deskryptora gniazda lub pliku

Interfejsy API gniazd

Wyszukiwanie funkcji API

Charakterystyka gniazd

Gniazda mają następujące cechy wspólne.

- Gniazdo jest reprezentowane przez liczbę całkowitą. Liczbę tę nazywa się *deskryptorem gniazda*.
- Gniazdo istnieje tak długo, jak długo proces utrzymuje otwarte połączenie z nim.
- Gniazdo można nazwać i wykorzystywać do komunikacji z innymi gniazdami w domenie komunikacyjnej.
- Gniazda się komunikują, gdy serwer przyjmuje od nich połączenia lub gdy wymienia z nimi komunikaty.
- 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 między programami. Program realizujący usługę (program serwera) uruchamia dostępne gniazdo, które może akceptować przychodzące żądania połączeń. Opcjonalnie serwer może przypisać udostępnianej usłudze nazwę, pozwalającą klientom na zidentyfikowanie miejsca, w którym usługa ta jest dostępna, i sposób połączenia się z nią. Klient usługi (program klienta) musi zażądać usługi od programu serwerowego. 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). Kiedy odbiorca wywołania (serwer, w tym przykładzie firma hydrauliczna) odbierze telefon, zostaje ustanowione połączenie. Hydraulik potwierdza, że dodzwoniono się do właściwej firmy, połączenie pozostaje więc aktywne tak długo, jak to potrzebne.

Komunikacja *bezpołączeniowa* zakłada, że nie jest ustanawiane żadne połączenie, przez które ma się odbywać dialog lub przesyłanie danych. Zamiast nawiązywać połączenie, serwer wyznacza nazwę identyfikującą miejsce, w którym jest osiągalny (można to porównać z numerem skrytki pocztowej). Kiedy wysyła się list na adres skrytki pocztowej, nie można mieć całkowitej pewności, że dotrze on do adresata. Zwykle należy poczekać na odpowiedź. Nie istnieje tu aktywne połączenie w czasie rzeczywistym, służące do wymiany danych.

Określanie parametrów gniazd

Kiedy aplikacja tworzy gniazdo za pomocą funkcji API socket(), musi zidentyfikować gniazdo za pomocą następujących parametrów:

- Rodzina adresów gniazda określa format struktury adresu gniazda. W tej sekcji przedstawiono przykłady struktury adresu dla każdej rodziny adresów.
- Typ gniazda określa formę komunikacji dla gniazda.
- Protokół gniazda określa obsługiwane protokoły używane przez gniazdo.

Parametry te definiują aplikację używającą gniazd oraz 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. W poniższej tabeli przedstawiono rodziny adresów oraz typy gniazd i protokoły z nimi powiązane:

Tabela 1. Podsumowanie parametrów 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_UNIX_CCSID	SOCK_STREAM	nie dotyczy

Tabela 1. Podsumowanie parametrów gniazd (kontynuacja)

Rodzina adresów	Typ gniazda	Protokół gniazda
	SOCK_DGRAM	nie dotyczy

Oprócz wymienionych parametrów gniazd w procedurach sieciowych i plikach nagłówkowych dostarczanych z biblioteką QSYSINC są zdefiniowane stałe wartości. Opisy plików nagłówkowych można znaleźć w opisie poszczególnych funkcji API. 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.

Odsyłacze pokrewne

"Funkcje sieciowe gniazd" na stronie 63
Funkcje sieciowe gniazd umożliwiają aplikacjom uzyskiwanie informacji od hostów, protokołów, usług oraz plików sieciowych.

Informacje pokrewne

Interfejsy API gniazd

Struktura adresu gniazda

Podczas przekazywania i odbierania adresów gniazda korzystają ze struktury adresu **sockaddr**. Struktura ta nie wymaga funkcji API gniazd do rozpoznawania formatu adresowania.

System operacyjny i5/OS obsługuje obecnie specyfikacje Berkeley Software Distribution (BSD) 4.3 oraz X/Open Single UNIX Specification (UNIX 98). Podstawowa funkcja API systemu i5/OS używa struktur i składni BSD 4.3. Poprzez zdefiniowanie dla makra _XOPEN_SOURCE wartości 520 lub większej można wybrać interfejs zgodny ze standardem UNIX 98. Dla każdej użytej struktury gniazd w specyfikacji BSD 4.3 istnieje odpowiednik w strukturze UNIX 98.

Tabela 2. Różnice w strukturach adresów między specyfikacjami BSD 4.3 a BSD 4.4/UNIX 98

Struktura BSD 4.3	Struktura BSD 4.4 zgodna ze standardem 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 adresu

Pole struktury adresu	Definicja
sa_len	Pole to zawiera długość adresu według specyfikacji UNIX 98. **Uwaga:** Pole **sa_len** udostępniono tylko w celu zapewnienia zgodności ze specyfikacją BSD 4.4. Nie ma jednak potrzeby używania tego pola nawet dla zapewnienia zgodności ze specyfikacjami BSD 4.4/UNIX 98. Pole jest ignorowane w przypadku adresów wejściowych.

Tabela 3. Struktura adresu (kontynuacja)

Pole struktury adresu	Definicja
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 zależy od typu 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	W tym polu jest deklarowany obszar pamięci masowej dla adresu z dowolnej rodziny adresów. Struktura ta jest wystarczająco duża i dopasowana do wszystkich struktur zależnych od użytego protokołu. Podczas korzystania z funkcji API można ją zatem rzutować jako strukturę as_sockaddr. Pole **ss_family** struktury sockaddr_storage jest zawsze zgodne z polem **family** w strukturze protokołu.

Rodzina adresów gniazd

Parametr rodziny adresów (address_family) w funkcji API socket() określa format struktury adresu, która będzie używana przez funkcje API gniazd.

Protokoły rodziny adresów zapewniają transport sieciowy danych aplikacji z jednej aplikacji do innej (lub z jednego procesu do innego w obrębie tego samego systemu). Aplikacja określa protokół transportu sieciowego w parametrze protokołu gniazda.

Rodzina adresów AF_INET

Ta rodzina adresów umożliwia komunikację między procesami działającymi w tym samym systemie lub w różnych systemach.

Adresy gniazd AF_INET składają się z adresu IP i numeru portu. Adres IP dla gniazda AF_INET można podawać w formacie adresu IP (na przykład 130.99.128.1) lub w formacie 32-bitowym (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 przestawiono w poniższej tabeli:

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

Struktura adresów sockaddr_in według specyfikacji BSD 4.3	Struktura adresów sockadrr_in według specyfikacji 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 według specyfikacji UNIX 98. **Uwaga:** Pole **sin_len** udostępniono tylko w celu zapewnienia zgodności ze specyfikacją BSD 4.4. Nie ma jednak potrzeby używania tego pola nawet dla zapewnienia zgodności ze specyfikacjami 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	Pole to zawiera adres IP.
sin_zero	Pole zastrzeżone. W polu należy wpisać szesnastkowe zera.

Odsyłacze pokrewne

"Korzystanie z rodziny adresów AF_INET" na stronie 27
Gniazda korzystające z rodziny adresów AF_INET mogą być zorientowane na połączenie (typ SOCK_STREAM) lub bezpołączeniowe (typ SOCK_DGRAM). Gniazda AF_INET zorientowane na połączenie używają jako protokołu transportowego TCP. Bezpołączeniowe gniazda AF_INET używają jako protokołu transportowego UDP.

Rodzina adresów AF_INET6

Ta rodzina adresów zapewnia obsługę protokołu IP 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 wartości szesnastkowe ośmiu 16-bitowych składników 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, jeśli do implementacji specyfikacji BSD 4.4/UNIX 98 zostanie użyte makro _XOPEN_SOURCE. W przypadku struktury adresów sockaddr_in6 różnice te przestawiono w poniższej tabeli:

Tabela 6. Różnice w strukturach adresów sockaddr_in między specyfikacjami BSD 4.3 a BSD 4.4/UNIX 98.

Struktura adresów sockaddr_in6 według specyfikacji BSD 4.3	Struktura adresów sockadrr_in6 według specyfikacji 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 według specyfikacji UNIX 98. **Uwaga:** Pole **sin6_len** udostępniono tylko w celu zapewnienia zgodności ze specyfikacją BSD 4.4. Nie ma jednak potrzeby używania tego pola nawet dla zapewnienia zgodności ze specyfikacjami BSD 4.4/UNIX 98. Pole jest ignorowane w przypadku adresów wejściowych.

Tabela 7. Struktura adresów AF_INET6 (kontynuacja)

Pole struktury adresu	Definicja
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:** Obecnie pole to nie jest obsługiwane, jego wartość należy więc ustawić na zero, tak aby zapewnić zgodność z przyszłymi wersjami.
sin6_addr	Pole to określa adres IPv6.
sin6_scope_id	Pole to identyfikuje zestaw interfejsów odpowiednich dla zakresu adresów określonych w polu **sin6_addr**.

Rodzina adresów AF_UNIX

Ta rodzina adresów umożliwia komunikację między procesami w ramach jednego systemu wykorzystującego 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 za pomocą funkcji API unlink().

Gniazda w ramach rodziny adresów AF_UNIX korzystają ze struktury adresów sockaddr_un. Struktura ta ulegnie zmianie, jeśli do implementacji specyfikacji BSD 4.4/UNIX 98 zostanie użyte makro _XOPEN_SOURCE. W odniesieniu do struktury adresów sockaddr_un różnice te podano w poniższej tabeli:

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

Struktura adresów sockaddr_un według specyfikacji BSD 4.3	Struktura adresów sockadrr un według specyfikacji 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 według specyfikacji UNIX 98. **Uwaga:** Pole **sun_len** udostępniono tylko w celu zapewnienia zgodności ze specyfikacją BSD 4.4. Nie ma jednak potrzeby używania tego pola nawet dla zapewnienia zgodności ze specyfikacjami 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. Mechanizm komunikacji między obydwoma procesami zależy od systemu.

Odsyłacze pokrewne

“Korzystanie z rodziny adresów AF_UNIX” na stronie 28
Gniazda rodziny adresów AF_UNIX lub AF_UNIX_CCSID mogą być zorientowane na połączenie (typ SOCK_STREAM) lub bezpołączeniowe (typ SOCK_DGRAM).

“Rodzina adresów AF_UNIX_CCSID”
Rodzina adresów AF_UNIX_CCSID jest zgodna z rodziną adresów AF_UNIX i ma takie same ograniczenia.

Informacje pokrewne

Funkcja API unlink() - usuwanie dowiązania do pliku

Rodzina adresów AF_UNIX_CCSID

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, a do łączności między procesami nie są wykorzystywane żadne zewnętrzne funkcje komunikacyjne. 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 umożliwia używanie nazwy ścieżek w kodzie UNICODE lub w dowolnym identyfikatorze CCSID poprzez zastosowanie formatu Qlg_Path_Name_T.

Ponieważ jednak gniazdo AF_UNIX może zwrócić nazwę ścieżki z gniazda AF_UNIX_CCSID w strukturze adresów AF_UNIX, długość 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 obrębie jednego gniazda. Jeśli podczas wywołania funkcji API socket() określi się rodzinę AF_UNIX_CCSID, to wszystkie adresy w późniejszych wywołaniach funkcji API muszą być określone 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 jest nią zawsze AF_UNIX_CCSID.
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 polu należy wpisać szesnastkowe zera.
sunc_qlg	Format nazwy ścieżki.

Tabela 10. Struktura adresu AF_UNIX_CCSID (kontynuacja)

Pole struktury adresu	Definicja
sunc_path	Nazwa ścieżki. Maksymalna długość nazwy wynosi 126 znaków i może być jedno- lub dwubajtowa. Nazwa może być zawarta w polu **sunc_path** lub przydzielana osobno i wskazywana przez wartość pola **sunc_path**. Format nazwy jest określony przez wartości pól **sunc_format** i **sunc_qlg**.

Odsyłacze pokrewne

“Korzystanie z rodziny adresów AF_UNIX_CCSID” na stronie 35
Gniazda rodziny adresów AF_UNIX_CCSID mają takie same specyfikacje jak gniazda rodziny adresów AF_UNIX. Gniazda rodziny adresów AF_UNIX_CCSID mogą być zorientowane na połączenie lub bezpołączeniowe. W celu zapewnienia komunikacji w tym samym systemie można stosować oba typy gniazd.

“Rodzina adresów AF_UNIX” na stronie 10
Ta rodzina adresów umożliwia komunikację między procesami w ramach jednego systemu wykorzystującego funkcje API gniazd. Adres jest w rzeczywistości nazwą ścieżki do pozycji systemu plików.

Informacje pokrewne

Format nazwy ścieżki

Typ gniazda

Drugi parametr w wywołaniu funkcji gniazda określa typ gniazda. Typ gniazda pozwala zidentyfikować typ oraz parametry połączenia uaktywnionego w celu przesyłania danych z jednego komputera lub procesu do drugiego.

System obsługuje następujące typy gniazd:

Strumieniowe (SOCK_STREAM)

Ten typ gniazd jest zorientowany na połączenie. Nawiązanie połączenia na całej trasie następuje za pomocą funkcji API bind(), listen(), accept() i connect(). SOCK_STREAM wysyła dane bez błędów czy powtórzeń i otrzymuje dane w kolejności wysyłania. W celu uniknięcia przekroczenia granicy danych SOCK_STREAM stosuje sterowanie przepływem. Nie narzuca granic bloków dla danych. Zakłada, że dane stanowią strumień bajtów. W implementacji i5/OS możliwe jest używanie gniazd strumieniowych w sieciach TCP, AF_UNIX i AF_UNIX_CCSID. Gniazd strumieniowych można także używać do komunikowania 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 gwarancji dostarczenia. Dane mogą zostać utracone lub zduplikowane. 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 przepływu przez sieć. Na gnieździe tego typu można wywoływać funkcję API connect(). Jednak w przypadku funkcji API connect() należy określić adres docelowy, pod który program wysyła dane i spod którego je odbiera. W implementacji i5/OS gniazd datagramowych można używać w sieciach UDP, AF_UNIX oraz AF_UNIX_CCSID.

Surowe (SOCK_RAW)

Gniazda tego typu umożliwiają bezpośredni dostęp do protokołów niższych warstw, takich jak IPv4 lub IPv6 oraz ICMP lub ICMP6. Typ SOCK_RAW wymaga większego doświadczenia programistycznego, gdyż trzeba zarządzać 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 gniazd realizują transport danych aplikacji przez sieć z jednego komputera na inny (lub z jednego procesu do innego procesu na tym samym komputerze).

Aplikacja określa protokół transportowy w parametrze **protocol** funkcji API socket().

Dla rodziny adresów AF_INET możliwe jest korzystanie z kilku protokołów transportowych. Protokoły architektury systemów sieciowych (SNA) i TCP/IP mogą być jednocześnie aktywne w ramach tego samego gniazda nasłuchującego. Atrybut sieciowy ALWANYNET (Zezwolenie na obsługę AnyNet) pozwala na wybranie dla aplikacji używających gniazd AF_INET transportu innego niż TCP/IP. Atrybut ten może przyjmować 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, dla atrybutu ALWANYNET powinien być ustawiony status *NO, tak aby bardziej efektywnie wykorzystać jednostkę centralną.

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

Gniazda AF_INET i AF_INET6 w sieci TCP/IP mogą być również typu SOCK_RAW, co oznacza, że komunikują się one bezpośrednio z warstwą sieci znaną jako IP (Internet Protocol). Zwykle z warstwą tą komunikują się protokoły TCP lub UDP. Przy korzystaniu z gniazd program użytkowy określa dowolny protokół z zakresu od 0 do 255 (z wyjątkiem protokołów TCP i UDP). Kiedy komputery komunikują się w sieci, ten numer protokołu jest następnie przesyłany w nagłówkach IP. Program użytkowy pełni zatem rolę protokołu transportowego, gdyż musi udostępnić wszystkie usługi, udostępniane normalnie przez protokoły UDP lub TCP.

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

Informacje pokrewne

Konfigurowanie APPC, APPN i HPR

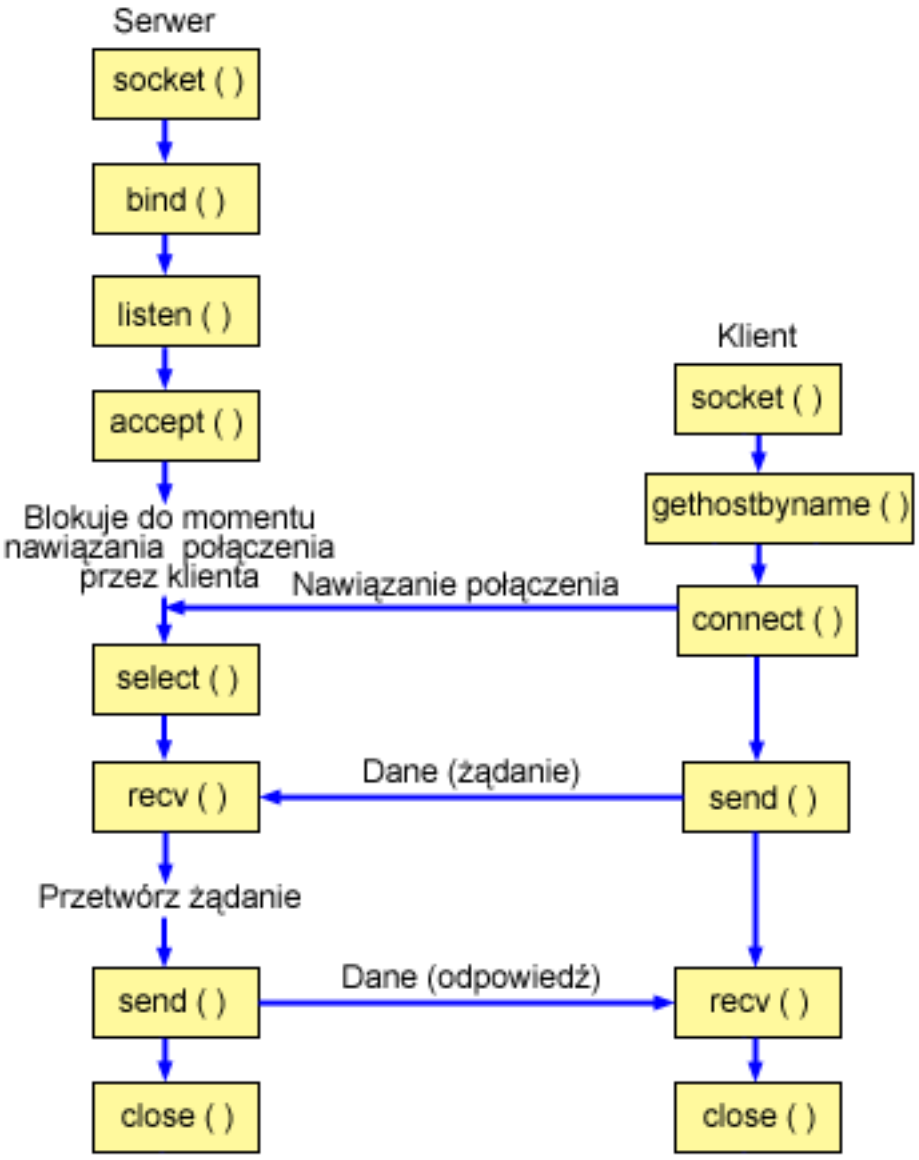
Podstawy projektowania gniazd

Przykłady te przedstawiają najpopularniejsze typy programów używających gniazd. Są to programy o najprostszej strukturze, które mogą jednak posłużyć do projektowania bardziej złożonych aplikacji używających gniazd.

Tworzenie gniazda zorientowanego na połączenie

W przykładowych programach serwera i klienta zilustrowano funkcje API gniazd napisane dla protokołu zorientowanego na połączenie, takiego jak TCP (Transmission Control Protocol).

Na poniższym rysunku zilustrowano relację klient/serwer funkcji API gniazd dla protokołu zorientowanego na połączenie.



Przebieg zdarzeń w gnieździe: serwer zorientowany na połączenie

Poniżej wyjaśniono sekwencję wywołań funkcji API gniazd, przedstawionych na rysunku. Opisano także relacje między aplikacją serwera a aplikacją klienta w architekturze zorientowanej na połączenie. Każdy zbiór przepływów zawiera odsyłacze do uwag dotyczących użycia poszczególnych funkcji API.

1. Funkcja API socket() zwraca deskryptor gniazda reprezentujący punkt końcowy. Instrukcja ta informuje również, że dla tego gniazda jest używana rodzina adresów AF_INET i protokół transportowy TCP (SOCK_STREAM).
2. Funkcja API setsockopt() umożliwia ponowne użycie adresu lokalnego w przypadku zrestartowania serwera przed upływem wymaganego czasu oczekiwania.
3. Po utworzeniu deskryptora gniazda funkcja API 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 API listen() pozwala serwerowi na przyjmowanie połączeń przychodzących od klientów. W tym przykładzie kolejka (backlog) ma wartość 10. Oznacza to, że system umieści w kolejce pierwsze 10 przychodzących żądań połączenia, kolejne zaś odrzuci.

5. Serwer akceptuje żądanie połączenia przychodzącego za pomocą funkcji API accept(). Wywołanie funkcji API accept() zostanie zablokowane na nieokreślony czas oczekiwania na połączenie przychodzące.
6. Funkcja API select() powoduje, że proces oczekuje na wystąpienie zdarzenia, po którym działanie zostaje aktywowane. W tym przykładzie system wysyła powiadomienie do procesu dopiero wtedy, gdy będą dostępne dane do odczytania. W wywołaniu funkcji API select() zastosowano 30-sekundowy limit czasu.
7. Funkcja API recv() odbiera dane z aplikacji klienta. W tym przykładzie klient wysyła 250 bajtów danych. W związku z tym można użyć opcji gniazda SO_RCVLOWAT, dzięki której funkcja API recv() pozostanie nieaktywna aż do nadejścia wszystkich 250 bajtów danych.
8. Funkcja API send() odsyła dane do klienta.
9. Funkcja API close() zamyka wszelkie otwarte deskryptory gniazd.

Przebieg zdarzeń w gnieździe: klient zorientowany na połączenie

Poniżej wyjaśniono sekwencję wywołań funkcji API, która opisuje relacje między aplikacją serwera a aplikacją klienta w architekturze zorientowanej na połączenie.

1. Funkcja API socket() zwraca deskryptor gniazda reprezentujący punkt końcowy. Instrukcja ta informuje również, że dla tego gniazda jest używana rodzina adresów AF_INET i protokół transportowy TCP (SOCK_STREAM).
2. W przykładowym programie klienckim, jeśli ciąg znaków serwera przesłany do funkcji API inet_addr() nie jest adresem IP w postaci liczb dziesiętnych oddzielonych kropkami, to zakłada się, że jest to nazwa hosta serwera. W takim przypadku do pobrania adresu IP serwera używa się funkcji API gethostbyname().
3. Po odebraniu deskryptora gniazda do nawiązania połączenia z serwerem należy użyć funkcji API connect().
4. Funkcja API send() wysyła na serwer 250 bajtów danych.
5. Funkcja API recv() oczekuje na odesłanie 250 bajtów danych z serwera. W tym przykładzie serwer odsyła otrzymane 250 bajtów danych. W przykładowym programie klienckim 250 bajtów danych może być przesyłane w oddzielnych pakietach, dlatego funkcji recv() można używać wielokrotnie, do momentu gdy zostanie odebrane całe 250 bajtów.
6. Funkcja API close() zamyka wszelkie otwarte deskryptory gniazd.

Informacje pokrewne

Funkcja API listen() - nasłuchiwanie przychodzących żądań połączenia

Funkcja API bind() - ustawianie lokalnego adresu gniazda

Funkcja API accept() - oczekiwanie na żądanie i nawiązywanie połączenia

Funkcja API send() - wysyłanie danych

Funkcja API recv() - odbieranie danych

Funkcja API close() - zamykanie deskryptora gniazda lub pliku

Funkcja API socket() - tworzenie gniazd

Funkcja API setsockopt_ioctl() - ustawianie opcji gniazd

Funkcja API select() - oczekiwanie na wydarzenia w wielu gniazdach

Funkcja API gethostbyname() - pobieranie informacji o hoście według nazwy hosta

Funkcja API connect() - nawiązywanie połączenia lub ustanawianie adresu docelowego

Przykład: program serwera zorientowanego na połączenie

W tym przykładzie przedstawiono sposób tworzenia programu serwera zorientowanego na połączenie.

Na podstawie tego programu przykładowego można tworzyć własne aplikacje serwera. Serwer zorientowany na połączenie jest jednym z najpowszechniejszych modeli aplikacji używających gniazd. W modelu zorientowanym na połączenie aplikacja serwera tworzy gniazdo, które służy do odbierania żądań od klientów.

Uwaga: Korzystając z przykładów, użytkownik wyraża zgodę na warunki podane w sekcji “Licencja na kod oraz Informacje dotyczące kodu” na stronie 199.

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

/*********************************************************************/
/* Pliki nagłówkowe wymagane przez program przykładowy                */
/*********************************************************************/
#include <stdio.h>
#include <sys/time.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

/*********************************************************************/
/* Stałe używane przez 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 reprezentujący punkt  */
      /* końcowy. Instrukcja ta informuje również, 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 funkcji socket()");
         break;
      }

      /*********************************************************************/
      /* Funkcja setsockopt() umożliwia ponowne użycie adresu lokalnego   */
      /* w razie zrestartowania serwera przed upływem wymaganego czasu    */
      /* oczekiwania.                                                      */
      /*********************************************************************/
      rc = setsockopt(sd, SOL_SOCKET, SO_REUSEADDR, (char *)&on, sizeof(on));
      if (rc < 0)
      {
         perror("Niepowodzenie funkcji 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 funkcji bind()");
   break;
}

/*********************************************************************/
/* Funkcja listen() pozwala serwerowi na przyjmowanie połączeń      */
/* przychodzących od klienta. W tym przykładzie kolejka (backlog)   */
/* ma wartość 10. Oznacza to, że system umieści w kolejce pierwsze  */
/* 10 przychodzących żądań połączenia,                               */
/* kolejne zaś odrzuci.                                              */
/*********************************************************************/
rc = listen(sd, 10);
if (rc < 0)
{
   perror("Niepowodzenie funkcji listen()");
   break;
}

printf("Gotowy do nawiązania połączenia z klientem.\n");

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

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

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

```
      }

      /*******************************************************************/
      /* W tym przykładzie wiadomo, że klient wyśle 250 bajtów danych.   */
      /* W związku z tym można użyć opcji gniazda SO_RCVLOWAT i określić, */
      /* że funkcja recv() ma pozostać nieaktywna aż do nadejścia        */
      /* wszystkich 250 bajtów danych.                                   */
      /*******************************************************************/
      length = BUFFER_LENGTH;
      rc = setsockopt(sd2, SOL_SOCKET, SO_RCVLOWAT,
                                        (char *)&length, sizeof(length));
      if (rc < 0)
      {
         perror("Niepowodzenie funkcji setsockopt(SO_RCVLOWAT)");
         break;
      }

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

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

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

   } while (FALSE);

   /*********************************************************************/
   /* Zamknij wszystkie otwarte deskryptory gniazd                      */
   /*********************************************************************/
   if (sd != -1)
      close(sd);
   if (sd2 != -1)
      close(sd2);
}
```

Odsyłacze pokrewne

“Korzystanie z rodziny adresów AF_INET” na stronie 27
Gniazda korzystające z rodziny adresów AF_INET mogą być zorientowane na połączenie (typ SOCK_STREAM) lub bezpołączeniowe (typ SOCK_DGRAM). Gniazda AF_INET zorientowane na połączenie używają jako protokołu transportowego TCP. Bezpołączeniowe gniazda AF_INET używają jako protokołu transportowego UDP.

Przykład: program klienta zorientowanego na połączenie

W tym przykładzie przedstawiono tworzenie programu klienta używającego gniazd, który umożliwia łączenie z serwerem zorientowanym na połączenie w systemie zorientowanym na połączenie.

Klient usługi (program klienta) musi zażądać usługi od programu serwerowego. Na podstawie tego przykładowego programu można tworzyć własne aplikacje klienta.

Uwaga: Korzystając z przykładów, użytkownik wyraża zgodę na warunki podane w sekcji “Licencja na kod oraz Informacje dotyczące kodu” na stronie 199.

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

/*********************************************************************/
/* Pliki nagłówkowe wymagane przez program przykładowy               */
/*********************************************************************/
#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 program                                       */
/*********************************************************************/
#define SERVER_PORT     3005
#define BUFFER_LENGTH    250
#define FALSE              0
#define SERVER_NAME     "ServerHostName"

/* Przekaż 1 parametr, będący albo         */
/* adresem, albo nazwą hosta serwera lub   */
/* ustaw nazwę serwera SERVER_NAME         */
/* w makrze #define                        */
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 reprezentujący punkt  */
      /* końcowy. Instrukcja ta informuje również, 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 funkcji 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 w postaci dziesiętnej z kropkami.         */
     /* Musi to zatem być nazwa hosta serwera. Do pobrania            */
     /* adresu IP serwera należy użyć funkcji 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, użyj funkcji connect().     */
  /*                                                                 */
  /*******************************************************************/
  rc = connect(sd, (struct sockaddr *)&serveraddr, sizeof(serveraddr));
  if (rc < 0)
  {
     perror("Niepowodzenie funkcji connect()");
     break;
  }

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

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

```
      /* W tym przykładzie wiadomo, że serwer odpowie wysłaniem tych      */
      /* samych 250 bajtów, które wysłano. 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 użyto innej metody. Ponieważ te 250       */
      /* bajtów danych może być przysłane w oddzielnych pakietach,        */
      /* funkcja recv() będzie uruchamiana wielokrotnie aż do             */
      /* nadejścia wszystkich 250 bajtów.                                 */
      /********************************************************************/
      bytesReceived = 0;
      while (bytesReceived < BUFFER_LENGTH)
      {
         rc = recv(sd, & buffer[bytesReceived],
                   BUFFER_LENGTH - bytesReceived, 0);
         if (rc < 0)
         {
            perror("Niepowodzenie funkcji recv()");
            break;
         }
         else if (rc == 0)
         {
            printf("Serwer zamknął połączenie\n");
            break;
         }

         /*****************************************************************/
         /* Zwiększ liczbę bajtów dotychczas otrzymanych                  */
         /*****************************************************************/
         bytesReceived += rc;
      }

   } while (FALSE);

   /***********************************************************************/
   /* Zamknij wszystkie otwarte deskryptory gniazd                        */
   /***********************************************************************/
   if (sd != -1)
      close(sd);
}
```

Odsyłacze pokrewne

“Korzystanie z rodziny adresów AF_INET” na stronie 27
Gniazda korzystające z rodziny adresów AF_INET mogą być zorientowane na połączenie (typ SOCK_STREAM) lub bezpołączeniowe (typ SOCK_DGRAM). Gniazda AF_INET zorientowane na połączenie używają jako protokołu transportowego TCP. Bezpołączeniowe gniazda AF_INET używają jako protokołu transportowego UDP.

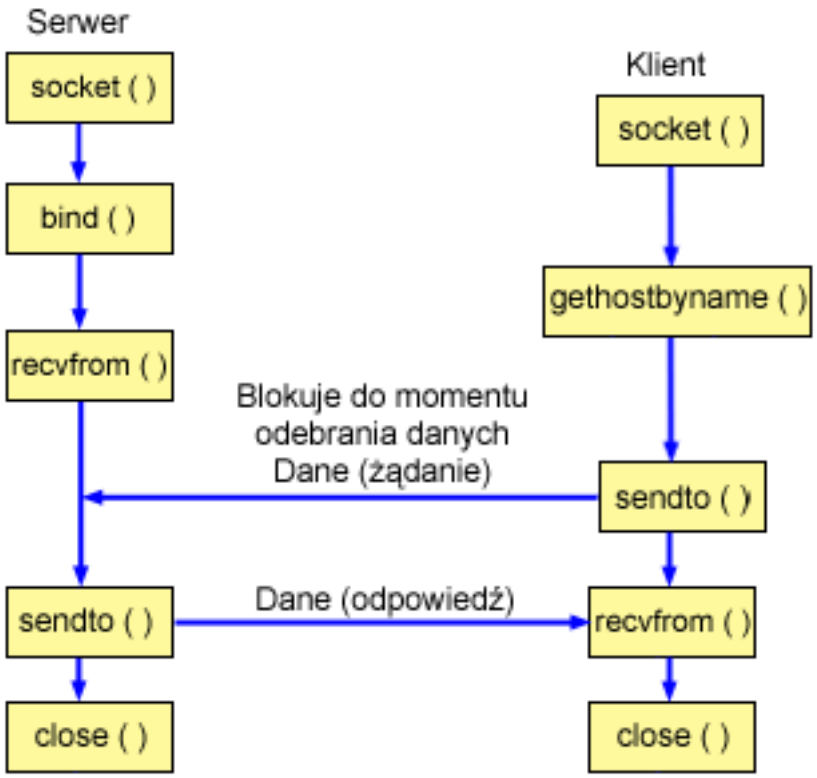
“Przykład: program serwera zorientowanego na połączenie” na stronie 15
W tym przykładzie przedstawiono sposób tworzenia programu serwera zorientowanego na połączenie.

Tworzenie gniazda bezpołączeniowego

Gniazda bezpołączeniowe nie nawiązują połączenia, przez które mogłyby być przesyłane dane. Zamiast tego aplikacja serwera określa nazwę, do której klient może wysyłać żądania.

Gniazda bezpołączeniowe zamiast protokołu TCP/IP używają protokołu UDP (User Datagram Protocol).

Na poniższym rysunku zilustrowano relację klient/serwer funkcji API gniazd użytych w przykładach dla protokołu bezpołączeniowego.



Przebieg zdarzeń w gnieździe: serwer bezpołączeniowy

Poniżej wyjaśniono sekwencję wywołań funkcji gniazd, przedstawionych na rysunku i w programach przykładowych. Opisano także relację między aplikacją serwera a aplikacją klienta w architekturze bezpołączeniowej. Każdy zbiór przepływów zawiera odsyłacze do uwag dotyczących użycia poszczególnych funkcji API. Więcej informacji na temat użycia tych funkcji API można uzyskać za pomocą wymienionych odsyłaczy. W pierwszym przykładzie serwer bezpołączeniowy używa następującej sekwencji wywołań funkcji API:

1. Funkcja API socket() zwraca deskryptor gniazda reprezentujący punkt końcowy. Instrukcja ta wskazuje również na to, że gniazdo używa rodziny adresów Internet Protocol (AF_INET) i protokołu transportowego UDP (SOCK_DGRAM).
2. Po utworzeniu deskryptora gniazda funkcja API bind() pobiera unikalną nazwę gniazda. W tym przykładzie użytkownik ustawia wartość s_addr na zero, co oznacza, że port UDP 3555 będzie powiązany ze wszystkimi adresami IPv4 w systemie.
3. Serwer odbierze te dane za pomocą funkcji API recvfrom(). Funkcja API recvfrom() oczekuje na przesłanie danych przez czas nieokreślony.
4. Funkcja API sendto() odsyła dane do klienta.
5. Funkcja API close() zamyka wszelkie otwarte deskryptory gniazd.

Przebieg zdarzeń w gnieździe: klient bezpołączeniowy

W drugim przykładzie klient bezpołączeniowy używa następującej sekwencji wywołań funkcji API.

1. Funkcja API socket() zwraca deskryptor gniazda reprezentujący punkt końcowy. Instrukcja ta wskazuje również na to, że gniazdo używa rodziny adresów Internet Protocol (AF_INET) i protokołu transportowego UDP (SOCK_DGRAM).
2. W przykładowym programie klienckim, jeśli ciąg znaków serwera przesłany do funkcji API inet_addr() nie jest adresem IP w postaci liczb dziesiętnych oddzielonych kropkami, to zakłada się, że jest to nazwa hosta serwera. W takim przypadku do pobrania adresu IP serwera używa się funkcji API gethostbyname().

3. Funkcja API sendto() odsyła dane do serwera.
4. Do odebrania danych z serwera służy funkcja API recvfrom().
5. Funkcja API close() zamyka wszelkie otwarte deskryptory gniazd.

Informacje pokrewne

Funkcja API close() - zamykanie deskryptora gniazda lub pliku

Funkcja API socket() - tworzenie gniazd

Funkcja API bind() - ustawianie lokalnego adresu gniazda

Funkcja API recvfrom() - odbieranie danych

Funkcja API sendto() - wysyłanie danych

Funkcja API gethostbyname() - pobieranie informacji o hoście według nazwy hosta

Przykład: program serwera bezpołączeniowego

W tym przykładzie przedstawiono sposób tworzenia programu serwera bezpołączeniowego używającego gniazd, korzystającego z protokołu UDP.

Uwaga: Korzystając z przykładów, użytkownik wyraża zgodę na warunki podane w sekcji “Licencja na kod oraz Informacje dotyczące kodu” na stronie 199.

```
/*********************************************************************/
/* Poniższy kod przykładowy dotyczy programu serwera bezpołączeniowego.   */
/*********************************************************************/

/*********************************************************************/
/* Pliki nagłówkowe wymagane przez program przykładowy                */
/*********************************************************************/
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

/*********************************************************************/
/* Stałe używane przez 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 reprezentujący punkt  */
      /* końcowy. Instrukcja ta informuje również, że dla tego gniazda    */
      /* zostanie użyta rodzina adresów INET (Internet Protocol)          */
      /* z protokołem transportowym UDP (SOCK_DGRAM).                     */
      /*********************************************************************/
```

```
      sd = socket(AF_INET, SOCK_DGRAM, 0);
      if (sd < 0)
    {
         perror("Niepowodzenie funkcji 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 funkcji 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 funkcji recvfrom()");
         break;
      }

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

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

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

   } while (FALSE);

   /*******************************************************************/
   /* Zamknij wszystkie otwarte deskryptory gniazd                    */
   /*******************************************************************/
   if (sd != -1)
      close(sd);
}
```

Przykład: program klienta bezpołączeniowego

W tym przykładzie przedstawiono użycie protokołu UDP do łączenia programu bezpołączeniowego klienta używającego gniazd z serwerem.

Uwaga: Korzystając z przykładów, użytkownik wyraża zgodę na warunki podane w sekcji “Licencja na kod oraz Informacje dotyczące kodu” na stronie 199.

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

/*********************************************************************/
/* Pliki nagłówkowe wymagane przez program przykładowy               */
/*********************************************************************/
#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 program                                       */
/*********************************************************************/
#define SERVER_PORT     3555
#define BUFFER_LENGTH    100
#define FALSE              0
#define SERVER_NAME     "ServerHostName"

/* Przekaż 1 parametr, będący albo         */
/* adresem, albo nazwą hosta serwera lub   */
/* ustaw nazwę serwera SERVER_NAME         */
/* w makrze #define                        */
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 reprezentujący punkt  */
      /* końcowy. Instrukcja ta informuje również, że dla tego gniazda    */
      /* zostanie użyta rodzina adresów INET (Internet Protocol)          */
      /* z protokołem transportowym TCP (SOCK_STREAM).                    */
      /*********************************************************************/
      sd = socket(AF_INET, SOCK_DGRAM, 0);
      if (sd < 0)
    {
         perror("Niepowodzenie funkcji 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 w postaci dziesiętnej z kropkami.         */
   /* Musi to zatem być nazwa hosta serwera. Do pobrania            */
   /* adresu IP serwera należy użyć funkcji 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 użyj funkcji sendto()               */
/*********************************************************************/
rc = sendto(sd, buffer, sizeof(buffer), 0,
            (struct sockaddr *)&serveraddr,
            sizeof(serveraddr));
if (rc < 0)
{
   perror("Niepowodzenie funkcji 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 funkcji 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);

   /*********************************************************************/
   /* Zamknij wszystkie otwarte deskryptory gniazd                      */
   /*********************************************************************/
   if (sd != -1)
      close(sd);
}
```

Projektowanie aplikacji używających rodzin adresów

W scenariuszach przedstawiono sposób projektowania aplikacji dla poszczególnych rodzin adresów, takich jak AF_INET, AF_INET6, AF_UNIX oraz AF_UNIX_CCSID.

Korzystanie z rodziny adresów AF_INET

Gniazda korzystające z rodziny adresów AF_INET mogą być zorientowane na połączenie (typ SOCK_STREAM) lub bezpołączeniowe (typ SOCK_DGRAM). Gniazda AF_INET zorientowane na połączenie używają jako protokołu transportowego TCP. Bezpołączeniowe gniazda AF_INET używają jako protokołu transportowego 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.

Odsyłacze pokrewne

"Rodzina adresów AF_INET" na stronie 8
Ta rodzina adresów umożliwia komunikację między procesami działającymi w tym samym systemie lub w różnych systemach.

"Wymagania wstępne dla programowania z użyciem gniazd" na stronie 2
Przed przystąpieniem do tworzenia aplikacji używających gniazd należy wykonać poniższe czynności, tak aby spełnione zostały wymagania dotyczące kompilatora, rodzin adresów AF_INET i AF_INET6 oraz funkcji API SSL i funkcji API z zestawu GSKit.

"Przykład: program serwera zorientowanego na połączenie" na stronie 15
W tym przykładzie przedstawiono sposób tworzenia programu serwera zorientowanego na połączenie.

"Przykład: program klienta zorientowanego na połączenie" na stronie 19
W tym przykładzie przedstawiono tworzenie programu klienta używającego gniazd, który umożliwia łączenie z serwerem zorientowanym na połączenie w systemie zorientowanym na połączenie.

Korzystanie z rodziny adresów AF_INET6

Gniazda AF_INET6 zapewniają obsługę 128-bitowych (16-bajtowych) struktur adresów protokołu IP w wersji 6 (IPv6). Programiści mogą tworzyć aplikacje korzystające z rodziny adresów AF_INET6, które będą akceptowały połączenia z węzłów klienckich obsługujących zarówno protokół IPv4, jak i IPv6 lub tylko protokół IPv6.

Podobnie jak gniazda AF_INET, gniazda AF_INET6 mogą być zorientowane na połączenie (typ SOCK_STREAM) lub bezpołączeniowe (typ SOCK_DGRAM). Gniazda AF_INET6 zorientowane na połączenie używają jako protokołu transportowego TCP. Bezpołączeniowe gniazda AF_INET6 używają jako protokołu transportowego 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.

Zgodność aplikacji IPv6 z aplikacjami IPv4

Aplikacje gniazd napisane dla rodziny adresów AF_INET6, używające protokołu IP wersja 6 (IPv6), współpracują z aplikacjami używającymi protokołu IP wersja 4 (IPv4), czyli korzystającymi z rodziny adresów AF_INET. Dzięki temu programiści tworzący aplikacje używające gniazd mogą korzystać z formatu adresu IPv6 odwzorowanego na adres IPv4. 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. Na przykład adres odwzorowany na adres IPv4 wygląda następująco:

```
::FFFF:192.1.1.1
```

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

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

Mimo że funkcja API 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 między gniazdem AF_INET6 a węzłami IPv4 i IPv6. Na przykład aplikacja wywołująca funkcję accept() na gnieździe nasłuchującym powiązanym ze strukturą in6addr_any będzie akceptowała połączenia zarówno z węzła IPv4, jak i IPv6. Zachowanie takie można modyfikować za pomocą opcji gniazd IPV6_V6ONLY na poziomie IPPROTO_IPV6. Nieliczne aplikacje będą musiały mieć dane na temat węzła, z którym współpracują. Dla takich aplikacji dostępne jest makro IN6_IS_ADDR_V4MAPPED() zdefiniowane w pliku <netinet/in.h>.

Odsyłacze pokrewne

“Wymagania wstępne dla programowania z użyciem gniazd” na stronie 2
Przed przystąpieniem do tworzenia aplikacji używających gniazd należy wykonać poniższe czynności, tak aby spełnione zostały wymagania dotyczące kompilatora, rodzin adresów AF_INET i AF_INET6 oraz funkcji API SSL i funkcji API z zestawu GSKit.

“Scenariusz dla gniazd: tworzenie aplikacji komunikujących się z klientami IPv4 i IPv6” na stronie 75
W tym przykładzie przedstawiono typową sytuację, w której można wykorzystać rodzinę adresów AF_INET6.

Informacje pokrewne

Porównanie protokołów IPv4 i IPv6

Funkcja API recvfrom() - odbieranie danych

Funkcja API accept() - oczekiwanie na żądanie i nawiązywanie połączenia

Funkcja API getpeername() - wczytywanie docelowego adresu gniazda

Funkcja API sendto() - wysyłanie danych

Funkcja API connect() - nawiązywanie połączenia lub ustanawianie adresu docelowego

Funkcja API bind() - ustawianie lokalnego adresu gniazda

Funkcja API gethostbyname() - pobieranie informacji o hoście według nazwy hosta

Funkcja API getaddrinfo() - pobieranie informacji o adresie

Funkcja API gethostbyaddr() - pobieranie informacji o hoście według adresu IP

Funkcja API getnameinfo() - pobieranie informacji o nazwie według adresu gniazda

Korzystanie z rodziny adresów AF_UNIX

Gniazda rodziny adresów AF_UNIX lub AF_UNIX_CCSID mogą być zorientowane na połączenie (typ SOCK_STREAM) lub bezpołączeniowe (typ SOCK_DGRAM).

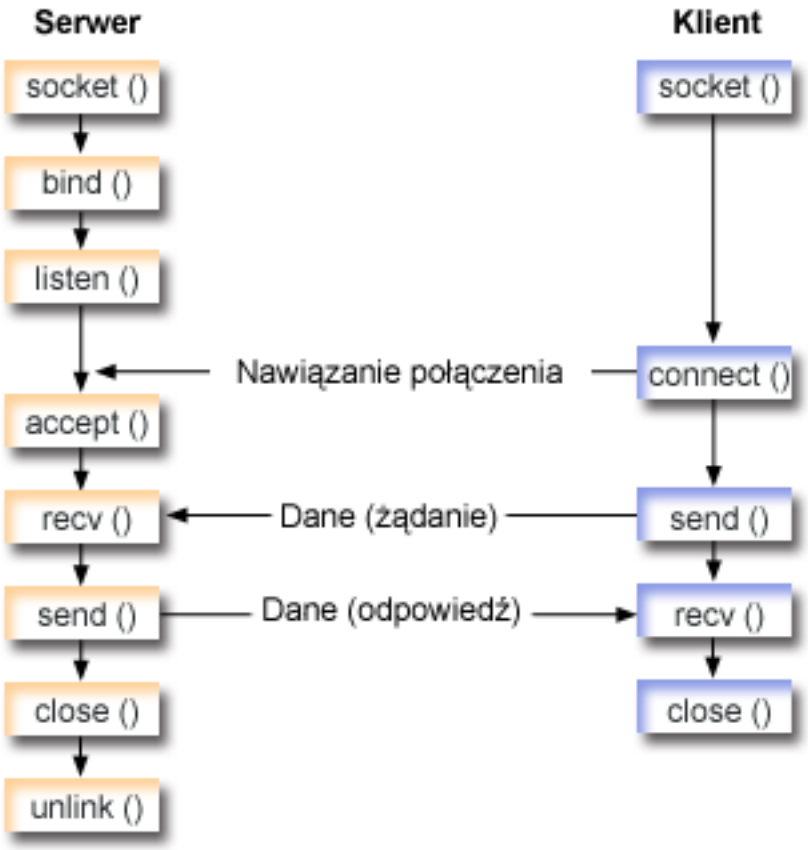
Ponieważ żadna zewnętrzna funkcja komunikacyjna nie łączy dwóch procesów, obydwa typy są tak samo niezawodne.

Gniazda datagramowe domeny systemu UNIX zachowują się inaczej niż gniazda datagramowe protokołu UDP. W przypadku gniazd datagramowych UDP program klienta nie musi wywoływać funkcji bind(), gdyż system automatycznie przypisze numer nieużywanego portu. Serwer może następnie odesłać datagram na ten numer portu. Należy jednak zaznaczyć, że w przypadku gniazd datagramowych domeny UNIX system nie przypisze automatycznie nazwy ścieżki dla klienta. Dlatego wszystkie programy klienckie korzystające z datagramów domeny UNIX muszą wywoływać funkcję API bind(). Dokładna nazwa ścieżki podana w funkcji bind() klienta jest wysyłana do serwera. Dlatego też, jeśli klient określi względną nazwę ścieżki (taką, 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, która nie jest pełna (nie zaczyna się od znaku /). Oznacza to, że położenie pozycji w hierarchii systemu plików powinno zostać określone względem bieżącego katalogu roboczego.

Uwaga: Nazwy ścieżek w systemie plików są obsługiwane przez narodowe wersje językowe.

Na poniższym rysunku zilustrowano relację klient/serwer rodziny adresów AF_UNIX.



Przebieg zdarzeń w gnieździe: aplikacja serwera używająca rodziny adresów AF_UNIX

W pierwszym przykładzie jest używana następująca sekwencja wywołań funkcji API:

1. Funkcja API socket() zwraca deskryptor gniazda reprezentujący punkt końcowy. Instrukcja ta informuje również, że dla tego gniazda zostanie użyta rodzina adresów UNIX z transportem strumieniowym (SOCK_STREAM). Gniazdo UNIX można również zainicjować za pomocą funkcji API socketpair().

Jedynymi rodzinami adresów obsługiwanymi przez funkcję API socketpair() są AF_UNIX i AF_UNIX_CCSID. Funkcja API socketpair() zwraca dwa nienazwane i podłączone deskryptory gniazd.

2. Po utworzeniu deskryptora gniazda funkcja API bind() pobiera unikalną nazwę gniazda.

   Obszar nazw dla gniazd domeny UNIX składa się z nazw ścieżek. Kiedy program używający gniazd wywołuje funkcję API bind(), tworzona jest pozycja w katalogu systemu plików. Jeśli ścieżka o podanej nazwie już istnieje, wywołanie funkcji API bind() kończy się niepowodzeniem. Dlatego też program używający gniazd domeny UNIX powinien zawsze wywołać funkcję API unlink(), aby po zakończeniu działania usunąć tę pozycję z katalogu.
3. Funkcja API listen() pozwala serwerowi na przyjmowanie połączeń przychodzących od klientów. W tym przykładzie kolejka (backlog) ma wartość 10. Oznacza to, że system umieści w kolejce pierwsze 10 przychodzących żądań połączenia, kolejne zaś odrzuci.
4. Funkcja API recv() odbiera dane z aplikacji klienta. W tym przykładzie klient wysyła 250 bajtów danych. W związku z tym można użyć opcji gniazda SO_RCVLOWAT, dzięki której funkcja API recv() pozostanie nieaktywna aż do nadejścia wszystkich 250 bajtów danych.
5. Funkcja API send() odsyła dane do klienta.
6. Funkcja API close() zamyka wszelkie otwarte deskryptory gniazd.
7. Funkcja API unlink() usuwa nazwę ścieżki UNIX z systemu plików.

Przebieg zdarzeń w gnieździe: aplikacja klienta używająca rodziny adresów AF_UNIX

W drugim przykładzie jest wykorzystywana następująca sekwencja wywołań funkcji API:

1. Funkcja API socket() zwraca deskryptor gniazda reprezentujący punkt końcowy. Instrukcja ta informuje również, że dla tego gniazda zostanie użyta rodzina adresów UNIX z transportem strumieniowym (SOCK_STREAM). Gniazdo UNIX można również zainicjować za pomocą funkcji API socketpair().

   Jedynymi rodzinami adresów obsługiwanymi przez funkcję API socketpair() są AF_UNIX i AF_UNIX_CCSID. Funkcja API socketpair() zwraca dwa nienazwane i podłączone deskryptory gniazd.
2. Po odebraniu deskryptora gniazda do nawiązania połączenia z serwerem należy użyć funkcji API connect().
3. Funkcja API send() wysyła 250 bajtów danych, określonych w aplikacji serwera za pomocą opcji SO_RCVLOWAT.
4. Funkcja API recv() wykonuje pętlę aż do momentu odebrania wszystkich 250 bajtów danych.
5. Funkcja API close() zamyka wszelkie otwarte deskryptory gniazd.

Odsyłacze pokrewne

“Rodzina adresów AF_UNIX” na stronie 10
Ta rodzina adresów umożliwia komunikację między procesami w ramach jednego systemu wykorzystującego funkcje API gniazd. Adres jest w rzeczywistości nazwą ścieżki do pozycji systemu plików.

“Wymagania wstępne dla programowania z użyciem gniazd” na stronie 2
Przed przystąpieniem do tworzenia aplikacji używających gniazd należy wykonać poniższe czynności, tak aby spełnione zostały wymagania dotyczące kompilatora, rodzin adresów AF_INET i AF_INET6 oraz funkcji API SSL i funkcji API z zestawu GSKit.

“Korzystanie z rodziny adresów AF_UNIX_CCSID” na stronie 35
Gniazda rodziny adresów AF_UNIX_CCSID mają takie same specyfikacje jak gniazda rodziny adresów AF_UNIX. Gniazda rodziny adresów AF_UNIX_CCSID mogą być zorientowane na połączenie lub bezpołączeniowe. W celu zapewnienia komunikacji w tym samym systemie można stosować oba typy gniazd.

Informacje pokrewne

Funkcja API close() - zamykanie deskryptora gniazda lub pliku

Funkcja API socket() - tworzenie gniazd

Funkcja API bind() - ustawianie lokalnego adresu gniazda

Funkcja API unlink() - usuwanie dowiązania do pliku

Funkcja API listen() - nasłuchiwanie przychodzących żądań połączenia

Funkcja API send() - wysyłanie danych

Funkcja API recv() - odbieranie danych

Funkcja API socketpair() - tworzenie pary gniazd

Funkcja API connect() - nawiązywanie połączenia lub ustanawianie adresu docelowego

Przykład: aplikacja serwera używająca rodziny adresów AF_UNIX:

W tym przykładzie przedstawiono program serwera używający rodziny adresów AF_UNIX. Rodzina adresów AF_UNIX używa wielu tych samych wywołań funkcji gniazd co inne rodziny adresów, przy czym do identyfikacji aplikacji serwera używa struktury nazwy ścieżki.

Uwaga: Korzystając z przykładów, użytkownik wyraża zgodę na warunki podane w sekcji “Licencja na kod oraz Informacje dotyczące kodu” na stronie 199.

```
/*********************************************************************/
/* Poniższy kod przykładowy przedstawia aplikację serwera używającą  */
/* rodziny adresów AF_UNIX                                           */
/*********************************************************************/

/*********************************************************************/
/* Pliki nagłówkowe wymagane przez program przykładowy               */
/*********************************************************************/
#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/un.h>

/*********************************************************************/
/* Stałe używane przez 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 reprezentujący punkt  */
      /* końcowy. Instrukcja ta informuje również, że dla tego gniazda    */
      /* zostanie użyta rodzina adresów UNIX_CCSID ze strumieniowym       */
      /* protokołem transportowym (SOCK_DGRAM).                           */
      /*********************************************************************/
      sd = socket(AF_UNIX, SOCK_STREAM, 0);
      if (sd < 0)
    {
         perror("Niepowodzenie funkcji 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 funkcji bind()");
   break;
}

/*******************************************************************/
/* Funkcja listen() pozwala serwerowi na przyjmowanie połączeń     */
/* przychodzących od klienta. W tym przykładzie kolejka (backlog)  */
/* ma wartość 10. Oznacza to, że system umieści w kolejce pierwsze */
/* 10 przychodzących żądań połączenia,                             */
/* kolejne zaś odrzuci.                                            */
/*******************************************************************/
rc = listen(sd, 10);
if (rc < 0)
{
   perror("Niepowodzenie funkcji listen()");
   break;
}

printf("Gotowy do nawiązania połączenia z klientem.\n");

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

/*******************************************************************/
/* W tym przykładzie wiadomo, że klient wyśle 250 bajtów danych.   */
/* W związku z tym można użyć opcji gniazda SO_RCVLOWAT i określić, */
/* że funkcja recv() ma pozostać nieaktywna aż do nadejścia        */
/* wszystkich 250 bajtów danych.                                   */
/*******************************************************************/
length = BUFFER_LENGTH;
rc = setsockopt(sd2, SOL_SOCKET, SO_RCVLOWAT,
                                   (char *)&length, sizeof(length));
if (rc < 0)
{
   perror("Niepowodzenie funkcji setsockopt(SO_RCVLOWAT)");
   break;
}
/*************************************************/
/* Odbierz 250 bajtów od klienta                 */
/*************************************************/
rc = recv(sd2, buffer, sizeof(buffer), 0);
if (rc < 0)
{
   perror("Niepowodzenie funkcji 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;
      }

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

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

   } while (FALSE);

   /*********************************************************************/
   /* Zamknij wszystkie otwarte deskryptory gniazd                      */
   /*********************************************************************/
   if (sd != -1)
      close(sd);

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

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

Przykład: aplikacja klienta używająca rodziny adresów AF_UNIX:

W tym przykładzie przedstawiono aplikację używającą rodziny adresów AF_UNIX w celu utworzenia połączenia klienta z serwerem.

Uwaga: Korzystając z przykładów, użytkownik wyraża zgodę na warunki podane w sekcji “Licencja na kod oraz Informacje dotyczące kodu” na stronie 199.

```
/*********************************************************************/
/* Poniższy kod przykładowy przedstawia aplikację klienta używającą  */
/* rodziny adresów AF_UNIX                                           */
/*********************************************************************/
/*********************************************************************/
/* Pliki nagłówkowe wymagane przez program przykładowy               */
/*********************************************************************/
#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/un.h>

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

/* Przekaż 1 parametr, będący albo        */
```

```
/* nazwą ścieżki serwera w postaci ciągu   */
/* znaków UNICODE, albo ścieżką serwera    */
/* ustawioną w makrze #define SERVER_PATH, */
/* 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 reprezentujący punkt */
      /* końcowy. Instrukcja ta informuje również, że dla tego gniazda   */
      /* zostanie użyta rodzina adresów UNIX_CCSID ze strumieniowym      */
      /* protokołem transportowym (SOCK_DGRAM).                          */
      /*******************************************************************/
      sd = socket(AF_UNIX, SOCK_STREAM, 0);
      if (sd < 0)
    {
         perror("Niepowodzenie funkcji 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, użyj funkcji connect().     */
      /*                                                                 */
      /*******************************************************************/
      rc = connect(sd, (struct sockaddr *)&serveraddr, SUN_LEN(&serveraddr));
      if (rc < 0)
      {
         perror("Niepowodzenie funkcji connect()");
         break;
      }

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

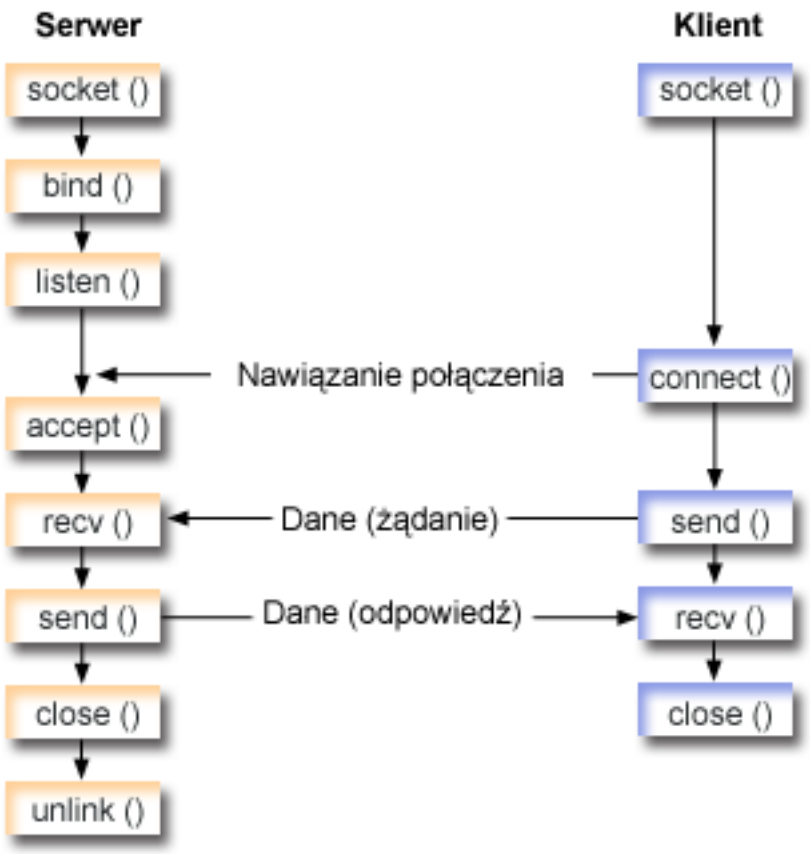
```
      /*******************************************************************/
      /* W tym przykładzie wiadomo, że serwer odpowie wysłaniem tych     */
      /* samych 250 bajtów, które wysłano. Ponieważ wiadomo, że          */
      /* zostanie odesłanych 250 bajtów, można użyć opcji gniazda   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

         /*******************************************************************/
         /* Zwiększ liczbę bajtów dotychczas otrzymanych                    */
         /*******************************************************************/
         bytesReceived += rc;
      }

   } while (FALSE);

   /*********************************************************************/
   /* Zamknij wszystkie otwarte deskryptory gniazd                      */
   /*********************************************************************/
   if (sd != -1)
      close(sd);
}
```

Korzystanie z rodziny adresów AF_UNIX_CCSID

Gniazda rodziny adresów AF_UNIX_CCSID mają takie same specyfikacje jak gniazda rodziny adresów AF_UNIX. Gniazda rodziny adresów AF_UNIX_CCSID mogą być zorientowane na połączenie lub bezpołączeniowe. W celu zapewnienia komunikacji w tym samym systemie można stosować oba typy gniazd.

Aby rozpocząć pracę z aplikacją używającą gniazd rodziny AF_UNIX_CCSID, należy zapoznać się ze strukturą Qlg_Path_Name_T i określić format wyjściowy.

Podczas pracy ze strukturą adresu wyjściowego, na przykład zwróconą przez funkcję API accept(), getsockname(), getpeername(), recvfrom() lub recvmsg(), aplikacja musi sprawdzić strukturę adresu gniazda (sockaddr_unc), aby określić jego format. Format wyjściowy nazwy ścieżki jest określany przez pola **sunc_format** i **sunc_qlg**. Należy jednak zaznaczyć, że gniazda nie zawsze używają tych samych wartości adresów wyjściowych co w adresach wejściowych.



Przebieg zdarzeń w gnieździe: aplikacja serwera używająca rodziny adresów AF_UNIX_CCSID

W pierwszym przykładzie jest używana następująca sekwencja wywołań funkcji API:

1. Funkcja API socket() zwraca deskryptor gniazda reprezentujący punkt końcowy. Instrukcja ta informuje również, że dla tego gniazda zostanie użyta rodzina adresów UNIX_CCSID z transportem strumieniowym (SOCK_STREAM). Gniazdo UNIX można również zainicjować za pomocą funkcji API socketpair().

   Jedynymi rodzinami adresów obsługiwanymi przez funkcję API socketpair() są AF_UNIX i AF_UNIX_CCSID. Funkcja API socketpair() zwraca dwa nienazwane i podłączone deskryptory gniazd.
2. Po utworzeniu deskryptora gniazda funkcja API bind() pobiera unikalną nazwę gniazda.

   Obszar nazw dla gniazd domeny UNIX składa się z nazw ścieżek. Kiedy program używający gniazd wywołuje funkcję API bind(), tworzona jest pozycja w katalogu systemu plików. Jeśli ścieżka o podanej nazwie już istnieje, wywołanie funkcji API bind() kończy się niepowodzeniem. Dlatego też program używający gniazd domeny UNIX powinien zawsze wywołać funkcję API unlink(), aby po zakończeniu działania usunąć tę pozycję z katalogu.
3. Funkcja API listen() pozwala serwerowi na przyjmowanie połączeń przychodzących od klientów. W tym przykładzie kolejka (backlog) ma wartość 10. Oznacza to, że system umieści w kolejce pierwsze 10 przychodzących żądań połączenia, kolejne zaś odrzuci.
4. Serwer akceptuje żądanie połączenia przychodzącego za pomocą funkcji API accept(). Wywołanie funkcji API accept() zostanie zablokowane na nieokreślony czas oczekiwania na połączenie przychodzące.
5. Funkcja API recv() odbiera dane z aplikacji klienta. W tym przykładzie klient wysyła 250 bajtów danych. W związku z tym można użyć opcji gniazda SO_RCVLOWAT, dzięki której funkcja API recv() pozostanie nieaktywna aż do nadejścia wszystkich 250 bajtów danych.
6. Funkcja API send() odsyła dane do klienta.
7. Funkcja API close() zamyka wszelkie otwarte deskryptory gniazd.

8. Funkcja API unlink() usuwa nazwę ścieżki UNIX z systemu plików.

Przebieg zdarzeń w gnieździe: aplikacja klienta używająca rodziny adresów AF_UNIX_CCSID

W drugim przykładzie jest wykorzystywana następująca sekwencja wywołań funkcji API:

1. Funkcja API socket() zwraca deskryptor gniazda reprezentujący punkt końcowy. Instrukcja ta informuje również, że dla tego gniazda zostanie użyta rodzina adresów UNIX z transportem strumieniowym (SOCK_STREAM). Gniazdo UNIX można również zainicjować za pomocą funkcji API socketpair().

   Jedynymi rodzinami adresów obsługiwanymi przez funkcję API socketpair() są AF_UNIX i AF_UNIX_CCSID. Funkcja API socketpair() zwraca dwa nienazwane i podłączone deskryptory gniazd.
2. Po odebraniu deskryptora gniazda do nawiązania połączenia z serwerem należy użyć funkcji API connect().
3. Funkcja API send() wysyła 250 bajtów danych, określonych w aplikacji serwera za pomocą opcji SO_RCVLOWAT.
4. Funkcja API recv() wykonuje pętlę aż do momentu odebrania wszystkich 250 bajtów danych.
5. Funkcja API close() zamyka wszelkie otwarte deskryptory gniazd.

Odsyłacze pokrewne

“Rodzina adresów AF_UNIX_CCSID” na stronie 11
Rodzina adresów AF_UNIX_CCSID jest zgodna z rodziną adresów AF_UNIX i ma takie same ograniczenia.

“Korzystanie z rodziny adresów AF_UNIX” na stronie 28
Gniazda rodziny adresów AF_UNIX lub AF_UNIX_CCSID mogą być zorientowane na połączenie (typ SOCK_STREAM) lub bezpołączeniowe (typ SOCK_DGRAM).

Informacje pokrewne

Format nazwy ścieżki

Funkcja API recvfrom() - odbieranie danych

Funkcja API accept() - oczekiwanie na żądanie i nawiązywanie połączenia

Funkcja API getpeername() - wczytywanie docelowego adresu gniazda

Funkcja API getsockname() - wczytywanie lokalnego adresu gniazda

Funkcja API recvmsg() - odbieranie komunikatów za pośrednictwem gniazda

Funkcja API close() - zamykanie deskryptora gniazda lub pliku

Funkcja API socket() - tworzenie gniazd

Funkcja API bind() - ustawianie lokalnego adresu gniazda

Funkcja API unlink() - usuwanie dowiązania do pliku

Funkcja API listen() - nasłuchiwanie przychodzących żądań połączenia

Funkcja API send() - wysyłanie danych

Funkcja API connect() - nawiązywanie połączenia lub ustanawianie adresu docelowego

Funkcja API recv() - odbieranie danych

Funkcja API socketpair() - tworzenie pary gniazd

Przykład: aplikacja serwera używająca rodziny adresów AF_UNIX_CCSID:

W tym programie przykładowym przedstawiono aplikację serwera używającą rodziny adresów AF_UNIX_CCSID.

Uwaga: Korzystając z przykładów, użytkownik wyraża zgodę na warunki podane w sekcji “Licencja na kod oraz Informacje dotyczące kodu” na stronie 199.

```
/*********************************************************************/
/* Poniższy kod przykładowy przedstawia aplikację serwera używającą  */
/* rodziny adresów AF_UNIX_CCSID.                                    */
/*********************************************************************/
```

```
/*********************************************************************/
/* Pliki nagłówkowe wymagane przez program przykładowy               */
/*********************************************************************/
#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/unc.h>

/*********************************************************************/
/* Stałe używane przez 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 reprezentujący punkt */
      /* końcowy. Instrukcja ta informuje również, że dla tego gniazda */
      /* zostanie użyta rodzina adresów UNIX_CCSID ze strumieniowym    */
      /* protokołem transportowym (SOCK_DGRAM).                       */
      /****************************************************************/
      sd = socket(AF_UNIX_CCSID, SOCK_STREAM, 0);
      if (sd < 0)
    {
         perror("Niepowodzenie funkcji 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 funkcji bind()");
         break;
      }

      /****************************************************************/
      /* Funkcja listen() pozwala serwerowi na przyjmowanie połączeń  */
```

```
/* przychodzących od klienta. W tym przykładzie kolejka (backlog)   */
/* ma wartość 10. Oznacza to, że system umieści w kolejce pierwsze  */
/* 10 przychodzących żądań połączenia,                              */
/* kolejne zaś odrzuci.                                             */
/********************************************************************/
rc = listen(sd, 10);
if (rc < 0)
{
   perror("Niepowodzenie funkcji listen()");
   break;
}

printf("Gotowy do nawiązania połączenia z klientem.\n");

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

/********************************************************************/
/* W tym przykładzie wiadomo, że klient wyśle 250 bajtów danych.    */
/* W związku z tym można użyć opcji gniazda SO_RCVLOWAT i określić, */
/* że funkcja recv() ma pozostać nieaktywna aż do nadejścia         */
/* wszystkich 250 bajtów danych.                                    */
/********************************************************************/
length = BUFFER_LENGTH;
rc = setsockopt(sd2, SOL_SOCKET, SO_RCVLOWAT,
                                    (char *)&length, sizeof(length));
if (rc < 0)
{
   perror("Niepowodzenie funkcji setsockopt(SO_RCVLOWAT)");
   break;
}


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

/********************************************************************/
/* Odeślij dane do klienta                                          */
/********************************************************************/
rc = send(sd2, buffer, sizeof(buffer), 0);
if (rc < 0)
{
```

```
         perror("Niepowodzenie funkcji send()");
         break;
      }

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

   } while (FALSE);

   /*********************************************************************/
   /* Zamknij wszystkie otwarte deskryptory gniazd                      */
   /*********************************************************************/
   if (sd != -1)
      close(sd);

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

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

Przykład: aplikacja klienta używająca rodziny adresów AF_UNIX_CCSID:

W tym programie przykładowym przedstawiono aplikację klienta używającą rodziny adresów AF_UNIX_CCSID.

Uwaga: Korzystając z przykładów, użytkownik wyraża zgodę na warunki podane w sekcji “Licencja na kod oraz Informacje dotyczące kodu” na stronie 199.

```
/*********************************************************************/
/* Poniższy kod przykładowy przedstawia aplikację klienta używającą  */
/* rodziny adresów AF_UNIX_CCSID.                                    */
/*********************************************************************/

/*********************************************************************/
/* Pliki nagłówkowe wymagane przez program przykładowy               */
/*********************************************************************/
#include <stdio.h>
#include <string.h>
#include <wcstr.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/unc.h>

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

/* Przekaż 1 parametr, będący albo         */
/* nazwą ścieżki serwera w postaci ciągu   */
/* znaków UNICODE, albo ścieżką serwera    */
/* ustawioną w makrze #define SERVER_PATH, */
/* 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 reprezentujący punkt*/
   /* końcowy. Instrukcja ta informuje również, że dla tego gniazda  */
   /* zostanie użyta rodzina adresów UNIX_CCSID ze strumieniowym     */
   /* protokołem transportowym (SOCK_DGRAM).                         */
   /******************************************************************/
   sd = socket(AF_UNIX_CCSID, SOCK_STREAM, 0);
   if (sd < 0)
 {
      perror("Niepowodzenie funkcji 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, użyj funkcji connect().    */
   /*                                                                */
   /******************************************************************/
   rc = connect(sd, (struct sockaddr *)&serveraddr, sizeof(serveraddr));
   if (rc < 0)
   {
      perror("Niepowodzenie funkcji connect()");
      break;
   }

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

```
      /*********************************************************************/
      /* W tym przykładzie wiadomo, że serwer odpowie wysłaniem tych      */
      /* samych 250 bajtów, które wysłano. 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 użyto innej metody. Ponieważ te 250       */
      /* bajtów danych może być przysłane w oddzielnych pakietach,        */
      /* funkcja recv() będzie uruchamiana wielokrotnie aż do             */
      /* nadejścia wszystkich 250 bajtów.                                 */
      /*********************************************************************/
      bytesReceived = 0;
      while (bytesReceived < BUFFER_LENGTH)
      {
         rc = recv(sd, & buffer[bytesReceived],
                   BUFFER_LENGTH - bytesReceived, 0);
         if (rc < 0)
         {
            perror("Niepowodzenie funkcji recv()");
            break;
         }
         else if (rc == 0)
         {
            printf("Serwer zamknął połączenie\n");
            break;
         }

         /*********************************************************************/
         /* Zwiększ liczbę bajtów dotychczas otrzymanych                      */
         /*********************************************************************/
         bytesReceived += rc;
      }

   } while (FALSE);

   /*********************************************************************/
   /* Zamknij wszystkie otwarte deskryptory gniazd                      */
   /*********************************************************************/
   if (sd != -1)
      close(sd);
}
```

Zaawansowane zagadnienia dotyczące gniazd

W poniższych sekcjach omówiono zaawansowane zagadnienia dotyczące gniazd, wykraczające poza zakres ogólnych wiadomości o ich istocie i działaniu. Zrozumienie tych zagadnień pozwala na projektowanie aplikacji używających gniazd, przeznaczonych do większych i bardziej złożonych sieci.

Asynchroniczne operacje we/wy

Funkcje API asynchronicznych operacji we/wy udostępniają metodę realizacji wątkowych modeli klient/serwer w celu zapewnienia wysoko współbieżnych operacji we/wy z efektywnym wykorzystaniem pamięci.

W poprzednich wątkowych modelach klient/serwer przeważały dwa modele operacji we/wy. W pierwszym z nich do każdego połączenia klienta jest przydzielony jeden wątek. Oznacza to jednak z nadmierną liczbę wątków i może powodować duże obciążenie związane z przejściem w stan nieaktywny i aktywowaniem. W modelu drugim liczbę wątków minimalizuje się dzięki użyciu funkcji API select() dla dużego zestawu połączeń klientów oraz przydzielaniu przygotowanego połączenia klienta lub żądania do wątku. W modelu tym należy wybierać lub zaznaczać wyniki wcześniejszych wyborów, co może wymagać wykonania znacznej, powtarzalnej pracy.

Asynchroniczne operacje we/wy i nakładanie operacji we/wy rozwiązują oba te problemy dzięki przesłaniu danych do i z buforu użytkownika po przekazaniu sterowania do aplikacji użytkownika. Asynchroniczne operacje we/wy wysyłają do wątków procesów roboczych powiadomienie o dostępności danych do odczytu oraz o gotowości połączenia do przesyłania danych.

Zalety asynchronicznych operacji we/wy

- Asynchroniczne operacje we/wy zarządzają zasobami systemowymi w sposób bardziej efektywny. Kopiowanie danych z i do buforów odbywa się asynchronicznie względem aplikacji inicjującej żądanie. To nakładające się przetwarzanie pozwala na efektywne wykorzystanie wielu procesorów i często przyspiesza stronicowanie, ponieważ po nadesłaniu danych bufory systemowe są zwalniane do ponownego wykorzystania.
- Asynchroniczne operacje we/wy zmniejszają czas oczekiwania dla procesu/wątku.
- Asynchroniczne operacje we/wy umożliwiają natychmiastowy dostęp do usługi na żądanie klienta.
- Asynchroniczne operacje we/wy zmniejszają przeciętne obciążenie przy przechodzeniu w stan nieaktywny i aktywowaniu.
- Asynchroniczne operacje we/wy zapewniają wydajną obsługę aplikacji impulsowych.
- Asynchroniczne operacje we/wy oferują większą skalowalność.
- Asynchroniczne operacje we/wy pozwalają na zastosowanie najbardziej efektywnej metody obsługi przesyłania danych o dużej objętości. Opcja fillBuffer funkcji API QsoStartRecv() informuje system operacyjny o uzyskaniu dużej ilości danych przed zakończeniem asynchronicznej operacji we/wy. Duże ilości danych można również przesyłać w ramach jednej operacji asynchronicznej.
- Asynchroniczne operacje we/wy zmniejszają liczbę potrzebnych wątków.
- Asynchroniczne operacje we/wy dają 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 operacje we/wy inicjuje chronioną sesję asynchronicznie przy użyciu funkcji API gsk_secure_soc_startInit().

Tabela 11. Funkcje API asynchronicznych operacji we/wy

Funkcja API	**Opis**
gsk_secure_soc_startInit()	Uruchamia asynchroniczne uzgadnianie sesji chronionej przy użyciu zestawu atrybutów dla środowiska SSL i sesji chronionej. **Uwaga:** Ta funkcja API obsługuje tylko gniazda typu SOCK_STREAM z rodziny adresów AF_INET lub AF_INET6.
gsk_secure_soc_startRecv()	Rozpoczyna operację asynchronicznego odbioru w ramach sesji chronionej. **Uwaga:** Ta funkcja API obsługuje tylko gniazda typu SOCK_STREAM z rodziny adresów AF_INET lub AF_INET6.
gsk_secure_soc_startSend()	Rozpoczyna operację asynchronicznego wysyłania w ramach sesji chronionej. **Uwaga:** Ta funkcja API obsługuje tylko gniazda typu SOCK_STREAM z rodziny adresów AF_INET lub AF_INET6.
QsoCreateIOCompletionPort()	Tworzy wspólny punkt oczekiwania dla zakończonych, nakładających się asynchronicznych operacji we/wy. Funkcja API QsoCreateIOCompletionPort() zwraca uchwyt portu reprezentującego punkt oczekiwania. Uchwyt ten należy podać w funkcjach API QsoStartRecv(), QsoStartSend(), QsoStartAccept(), gsk_secure_soc_startRecv() lub gsk_secure_soc_startSend() , tak aby zainicjować nakładające się, asynchroniczne operacje we/wy. Ponadto uchwyt ten może zostać użyty z funkcją QsoPostIOCompletion() w celu przesłania zdarzenia do powiązanego portu zakończenia operacji we/wy.

Tabela 11. Funkcje API asynchronicznych operacji we/wy (kontynuacja)

Funkcja API	Opis
QsoDestroyIOCompletionPort()	Niszczy port zakończenia operacji we/wy.
QsoWaitForIOCompletionPort()	Czeka na zakończone, nakładające się operacje we/wy. Port zakończenia operacji we/wy reprezentuje ten port oczekiwania.
QsoStartAccept()	Rozpoczyna asynchroniczną operację akceptowania. **Uwaga:** Ta funkcja API obsługuje tylko gniazda typu SOCK_STREAM z rodziny adresów AF_INET lub AF_INET6.
QsoStartRecv()	Rozpoczyna asynchroniczną operację odbioru. **Uwaga:** Ta funkcja API obsługuje tylko gniazda typu SOCK_STREAM z rodziny adresów AF_INET lub 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 API lub aktywności.

Sposób działania asynchronicznych operacji we/wy

Aplikacja tworzy port zakończenia operacji we/wy 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 przez podanie uchwytu portu zakończenia operacji we/wy. Kiedy operacje we/wy zostają zakończone, informacje o statusie i uchwyt zdefiniowany przez aplikację są zapisywane w podanym porcie zakończenia operacji we/wy. Zapisanie do portu zakończenia operacji we/wy uaktywnia dokładnie jeden z wielu możliwych wątków oczekujących. Aplikacja odbiera następujące dane:

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

Ten uchwyt aplikacji może być deskryptorem gniazda, wskazującym połączenie klienta, lub wskaźnikiem pamięci, która zawiera dodatkowe informacje o stanie połączenia klienta. Ponieważ operacja się zakończyła, 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 obsłużyć wiele różnych żądań klienta i nie są powiązane z tylko jednym z nich. Ponieważ kopiowanie z i do buforów użytkowników odbywa się asynchronicznie względem procesów serwera, czasy oczekiwania dla żądań klientów są krótsze. Cecha ta może być szczególnie korzystna w systemach wieloprocesorowych.

Struktura asynchronicznych operacji we/wy

Aplikacja korzystająca z asynchronicznych operacji we/wy ma strukturę przedstawioną w przykładowym fragmencie kodu:

```
#include <qsoasync.h>
struct Qso_OverlappedIO_t
{
  Qso_DescriptorHandle_t descriptorHandle;
  void *buffer;
  size_t  bufferLength;
  int postFlag : 1;
  int fillBuffer : 1;
  int postFlagResult : 1;
  int reserved1 : 29;
  int returnValue;
  int errnoValue;
```

```
  int operationCompleted;
  int secureDataTransferSize;
  unsigned int bytesAvailable;
  struct timeval operationWaitTime;
  int postedDescriptor;
  char reserved2[40];
}
```

Odsyłacze pokrewne

“Przykład: korzystanie z asynchronicznych operacji we/wy” na stronie 112
Aplikacja tworzy port zakończenia operacji we/wy za pomocą funkcji API QsoCreateIOCompletionPort(). Funkcja ta zwraca uchwyt, za pomocą którego można zaplanować asynchroniczne żądania we/wy i oczekiwać na ich zakończenie.

“Zalecenia dotyczące projektowania aplikacji używających gniazd” na stronie 84
Przed przystąpieniem do pracy z aplikacją używającą gniazd należy ocenić jej wymagania funkcjonalne, cele i potrzeby. Należy również rozważyć wymagania aplikacji dotyczące wydajności oraz jej wpływ na zasoby systemu.

“Przykłady: projekty aplikacji zorientowanych na połączenie” na stronie 87
System udostępnia kilka metod projektowania serwera używającego gniazd zorientowanego na połączenia. Do tworzenia własnych programów zorientowanych na połączenie można użyć poniższych programów przykładowych.

“Przykład: używanie sygnałów z blokującymi funkcjami API gniazd” na stronie 168
Sygnały mogą przekazywać powiadomienie o tym, że doszło do zablokowania procesu lub aplikacji. Udostępniają także limit czasu blokowania procesów.

Informacje pokrewne

Funkcja API gsk_secure_soc_startInit() - uruchamianie operacji asynchronicznej w celu uzgadniania sesji chronionej

Funkcja API gsk_secure_soc_startRecv() - uruchamianie asynchronicznej operacji odbierania podczas sesji chronionej

Funkcja API gsk_secure_soc_startSend() - uruchamianie asynchronicznej operacji wysyłania podczas sesji chronionej

Funkcja API QsoCreateIOCompletionPort() - tworzenie portu zakończenia operacji we/wy

Funkcja API QsoDestroyIOCompletionPort() - likwidacja portu zakończenia operacji we/wy

Funkcja API QsoWaitForIOCompletion() - oczekiwanie na operację we/wy

Funkcja API QsoStartAccept() - uruchamianie asynchronicznej operacji akceptowania

Funkcja API QsoStartSend() - uruchamianie asynchronicznej operacji wysyłania

Funkcja API QsoStartRecv() - uruchamianie asynchronicznej operacji odbierania

Funkcja API QsoPostIOCompletion() - zapisywanie żądania zakończenia operacji we/wy

Gniazda chronione

Aplikacje używające gniazd chronionych w systemie operacyjnym i5/OS można obecnie tworzyć dwiema metodami. Funkcje API SSL_ i GSKit (Global Secure Toolkit) zapewniają prywatność komunikacji w otwartych sieciach komunikacyjnych, czyli najczęściej w Internecie.

Funkcje te pozwalają aplikacjom typu klient/serwer na komunikowanie się w taki sposób, aby uniemożliwić podsłuch przekazu, manipulowanie nim i fałszowanie go. Zarówno funkcje API SSL_, jak i API GSKit obsługują uwierzytelnianie serwera i klienta oraz umożliwiają aplikacjom korzystanie z protokołu SSL. Funkcje API GSKit są jednak obsługiwane we wszystkich systemach IBM, podczas gdy funkcje API SSL_ występują tylko w systemie operacyjnym i5/OS. Aby zwiększyć przenośność między systemami, należy przy tworzeniu aplikacji obsługujących połączenia przez gniazda chronione korzystać raczej z funkcji API GSKit.

Przegląd gniazd chronionych

Protokół SSL (Secure Sockets Layer), opracowany przez firmę Netscape, jest protokołem warstwowym, przeznaczonym do użytku w połączeniu z niezawodnym protokołem transportowym, takim jak Transmission Control

Protocol (TCP), w celu zapewnienia aplikacjom bezpiecznej komunikacji. Jest ona niezbędna w wielu zastosowaniach, na przykład w komunikacji z użyciem protokołów HTTP, FTP, SMTP i Telnet.

Aplikacja korzystająca z protokołu SSL musi na ogół używać innego portu niż aplikacja, która nie korzysta z SSL. Na przykład przeglądarka obsługująca protokół SSL łączy się z serwerem HTTP obsługującym SSL za pomocą adresu URL zaczynającego się od https, a nie http. W większości przypadków adres URL z przedrostkiem https próbuje otworzyć połączenie z serwerem na porcie 443, a nie na porcie 80, którego używa standardowy serwer HTTP.

Istnieje wiele wersji protokołu SSL. Najnowsza, Transport Layer Security (TLS) wersja 1.0, stanowi ewolucyjną aktualizację protokołu SSL wersja 3.0. Zarówno funkcje API SSL_, jak i 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 oraz SSL wersja 3.0 kompatybilna z wersją 2.0. Więcej informacji na temat TLS wersja 1.0 można znaleźć w dokumencie RFC 2246: ″The TLS Protocol Version 1.0″ .

Funkcje API z zestawu Global Secure ToolKit (GSKit)

GSKit to zestaw programowalnych interfejsów, który umożliwia aplikacjom obsługę warstwy SSL.

Funkcje API GSKit, podobnie jak funkcje SSL_, umożliwiają implementowanie protokołów SSL i TLS w aplikacjach używających gniazd. Należy jednak zaznaczyć, że funkcje API GSKit są obsługiwane we wszystkich systemach IBM. Co więcej, programowanie za ich pomocą jest łatwiejsze niż w przypadku funkcji SSL_. W zestawie GSKit wprowadzono ponadto nowe funkcje API, które umożliwiają asynchroniczne uzgadnianie sesji chronionej oraz wysyłanie i odbieranie chronionych danych. Te asynchroniczne funkcje API istnieją tylko w systemie operacyjnym i5/OS. Nie można ich przenieść do innych systemów.

Uwaga: Funkcje API GSKit obsługują tylko gniazda typu SOCK_STREAM z rodziny adresów AF_INET i AF_INET6.

W poniższej tabeli przedstawiono opis funkcji API GSKit.

Tabela 12. Funkcje API z zestawu Global Secure Toolkit

Funkcja API	Opis
gsk_attribute_get_buffer()	Uzyskuje specyficzne informacje w postaci łańcucha znaków o sesji chronionej 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 sesji chronionej lub środowiska SSL.
gsk_attribute_get_enum_value()	Uzyskuje wartości wyszczególnionych danych dla sesji chronionej lub środowiska SSL.
gsk_attribute_get_numeric_value()	Uzyskuje konkretne informacje numeryczne o sesji chronionej lub środowisku SSL.
gsk_attribute_set_callback()	Ustawia wskaźniki wywołań zwrotnych do procedur w aplikacji użytkownika. Użytkownik może skorzystać z tych procedur do szczególnych celów.
gsk_attribute_set_buffer()	Nadaje określonemu atrybutowi buforu wartość odpowiadającą określonej sesji chronionej lub środowisku SSL.
gsk_attribute_set_enum()	Przypisuje wyszczególniony atrybut typu do wartości wyszczególnionej w sesji chronionej lub środowisku SSL.
gsk_attribute_set_numeric_value()	Przypisuje określone informacje numeryczne sesji chronionej lub środowisku SSL.
gsk_environment_close()	Zamyka środowisko SSL i zwalnia całą pamięć przypisaną środowisku.

Tabela 12. Funkcje API z zestawu Global Secure Toolkit (kontynuacja)

Funkcja API	Opis
gsk_environment_init()	Inicjuje środowisko SSL po ustawieniu wszystkich 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 sesję chronioną i zwalnia wszystkie zasoby jej przypisane.
gsk_secure_soc_init()	Uzgadnia sesję chronioną przy użyciu zestawu atrybutów dla środowiska SSL i sesji chronionej.
gsk_secure_soc_misc()	Wykonuje różne funkcje API związane z sesją chronioną.
gsk_secure_soc_open()	Uzyskuje pamięć dla sesji chronionej, ustawia domyślne wartości atrybutów oraz zwraca uchwyt, który musi zostać zapisany i użyty podczas kolejnych wywołań funkcji API związanych z sesją chronioną.
gsk_secure_soc_read()	Odbiera dane podczas sesji chronionej.
gsk_secure_soc_startInit()	Uruchamia asynchroniczne uzgadnianie sesji chronionej przy użyciu zestawu atrybutów dla środowiska SSL i sesji chronionej.
gsk_secure_soc_write()	Zapisuje dane podczas sesji chronionej.
gsk_secure_soc_startRecv()	Inicjuje asynchroniczną operację odbioru podczas sesji chronionej.
gsk_secure_soc_startSend()	Inicjuje asynchroniczną operację wysyłania podczas sesji chronionej.
gsk_strerror()	Pobiera komunikat o błędzie i powiązany łańcuch tekstowy, który opisuje wartość zwracaną podczas wywołania funkcji API GSKit.

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

1. Wywołanie funkcji API socket() w celu uzyskania deskryptora gniazda.
2. Wywołanie funkcji API gsk_environment_open() w celu uzyskania uchwytu środowiska SSL.
3. Co najmniej jedno wywołanie funkcji API gsk_attribute_set_xxxxx() w celu ustawienia atrybutów środowiska SSL. Minimum to wywołanie funkcji API 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. Należy ponadto ustawić typ aplikacji (klient lub serwer), czyli wartość GSK_SESSION_TYPE, przy użyciu funkcji API gsk_attribute_set_enum().
4. Wywołanie funkcji API gsk_environment_init() w celu zainicjowania tego środowiska do przetwarzania SSL i określenia informacji o bezpieczeństwie dla wszystkich sesji SSL, które będą uruchamiane w tym środowisku.
5. Wywołania gniazd w celu uaktywnienia połączenia. Aplikacja wywołuje funkcję API connect() w celu uaktywnienia połączenia z programem klienckim lub funkcje API bind(), listen() i accept() w celu umożliwienia serwerowi akceptowania przychodzących żądań połączeń.
6. Wywołanie funkcji API gsk_secure_soc_open() w celu uzyskania uchwytu dla sesji chronionej.
7. Co najmniej jedno wywołanie funkcji API gsk_attribute_set_xxxxx() w celu ustawienia atrybutów sesji chronionej. Minimum to wywołanie funkcji API gsk_attribute_set_numeric_value() w celu powiązania określonego gniazda z tą sesją chronioną.
8. Wywołanie funkcji gsk_secure_soc_init() w celu zainicjowania negocjacji parametrów szyfrowania podczas uzgadniania SSL.

Uwaga: Zwykle aby uzgadnianie SSL się powiodło, program serwera musi okazać certyfikat. Serwer musi ponadto mieć dostęp do klucza prywatnego powiązanego z certyfikatem serwera i do zbioru bazy danych kluczy, w którym jest przechowywany certyfikat. W niektórych przypadkach także klient musi okazać certyfikat podczas przetwarzania uzgadniania SSL. Dotyczy to sytuacji, gdy na serwerze, z którym łączy się klient, włączono uwierzytelnianie klienta. Wywołania funkcji API gsk_attribute_set_buffer (GSK_OS400_APPLICATION_ID) lub gsk_attribute_set_buffer (GSK_KEYRING_FILE) identyfikują (w różny sposób) zbiór bazy danych kluczy, za pomocą którego uzyskano certyfikat i klucz prywatny użyte podczas uzgadniania.

9. Wywołania funkcji API gsk_secure_soc_read() i gsk_secure_soc_write() w celu odbierania i wysyłania danych.
10. Wywołanie funkcji API gsk_secure_soc_close() w celu zakończenia sesji chronionej.
11. Wywołanie funkcji API gsk_environment_close() w celu zamknięcia środowiska SSL.
12. Wywołanie funkcji API close() w celu usunięcia podłączonego gniazda.

Odsyłacze pokrewne

“Przykład: chroniony serwer GSKit z asynchronicznym odbieraniem danych” na stronie 119
W tym przykładzie przedstawiono sposób ustanawiania chronionego serwera za pomocą funkcji API GSKit.

“Przykład: chroniony serwer GSKit z uzgadnianiem asynchronicznym” na stronie 129
Funkcja API gsk_secure_soc_startInit() umożliwia tworzenie chronionych aplikacji serwera, które obsługują żądania w sposób asynchroniczny.

“Przykład: ustanawianie chronionego klienta za pomocą funkcji API Global Secure ToolKit” na stronie 139
W tym przykładzie przedstawiono sposób ustanawiania klienta za pomocą funkcji API GSKit.

Informacje pokrewne

Funkcja API gsk_attribute_get_buffer() - uzyskiwanie informacji znakowych o sesji chronionej lub środowisku SSL

Funkcja API gsk_attribute_get_cert_info() - uzyskiwanie informacji o certyfikacie lokalnym lub certyfikacie partnera

Funkcja API gsk_attribute_get_enum() - uzyskiwanie informacji wyliczeniowych o sesji chronionej lub środowisku SSL

Funkcja API gsk_attribute_get_numeric_value() - uzyskiwanie informacji liczbowych o sesji chronionej lub środowisku SSL

Funkcja API gsk_attribute_set_callback() - ustawianie wskaźników wywołania zwrotnego do procedur w aplikacjach użytkownika

Funkcja API gsk_attribute_set_buffer() - ustawianie informacji znakowych dla sesji chronionej lub środowiska SSL

Funkcja API gsk_attribute_set_enum() - ustawianie informacji wyliczeniowych dla sesji chronionej lub środowiska SSL

Funkcja API gsk_attribute_set_numeric_value() - ustawianie informacji liczbowych dla sesji chronionej lub środowiska SSL

Funkcja API gsk_environment_close() - zamykanie środowiska SSL

Funkcja API gsk_environment_init() - inicjowanie środowiska SSL

Funkcja API gsk_environment_open() - pobierania uchwytu dla środowiska SSL

Funkcja API gsk_secure_soc_close() - zamykanie sesji chronionej

Funkcja API gsk_secure_soc_init() - uzgadnianie sesji chronionej

Funkcja API gsk_secure_soc_misc() - wykonywanie różnych działań w sesji chronionej

Funkcja API gsk_secure_soc_open() - pobieranie uchwytu dla sesji chronionej

Funkcja API gsk_secure_soc_startInit() - uruchamianie operacji asynchronicznej w celu uzgadniania sesji chronionej

Funkcja API gsk_secure_soc_read() - odbieranie danych podczas sesji chronionej

Funkcja API gsk_secure_soc_write() - wysyłanie danych podczas sesji chronionej

Funkcja API gsk_secure_soc_startRecv() - uruchamianie asynchronicznej operacji odbierania podczas sesji chronionej

Funkcja API gsk_secure_soc_startSend() - uruchamianie asynchronicznej operacji wysyłania podczas sesji chronionej

Funkcja API gsk_strerror() - odczytywanie komunikatu o błędzie wykonania GSKit

Funkcja API socket() - tworzenie gniazd

Funkcja API bind() - ustawianie lokalnego adresu gniazda

Funkcja API connect() - nawiązywanie połączenia lub ustanawianie adresu docelowego

Funkcja API listen() - nasłuchiwanie przychodzących żądań połączenia

Funkcja API accept() - oczekiwanie na żądanie i nawiązywanie połączenia

Funkcja API close() - zamykanie deskryptora gniazda lub pliku

Funkcje API SSL_

Funkcje API SSL_ umożliwiają tworzenie aplikacji używających gniazd chronionych w systemie operacyjnym i5/OS.

W przeciwieństwie do funkcji API GSKit funkcje API SSL_ występują tylko w systemie i5/OS. W poniższej tabeli przedstawiono opis funkcji API SSL_ obsługiwanych w implementacji i5/OS.

*Tabela 13. Funkcje API SSL_*

Funkcja API	**Opis**
SSL_Create()	Włącza obsługę SSL dla określonego deskryptora gniazda.
SSL_Destroy()	Kończy obsługę SSL dla określonej sesji i danego gniazda SSL.
SSL_Handshake()	Inicjuje protokół uzgadniania SSL.
SSL_Init()	Inicjuje bieżące zadanie dla SSL i określa informacje o bezpieczeństwie dla bieżącego zadania. **Uwaga:** Aby użyć protokołu SSL, należy wcześniej uruchomić funkcję API SSL_Init() lub SSL_Init_Application().
SSL_Init_Application()	Inicjuje bieżące zadanie dla SSL i określa informacje o bezpieczeństwie dla bieżącego zadania. **Uwaga:** Aby użyć protokołu SSL, należy wcześniej 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()	Wczytuje komunikaty o błędach wykonania w SSL.
SSL_Perror()	Drukuje komunikaty o błędach SSL.
QlgSSL_Init()	Inicjuje bieżące zadanie dla SSL i określa informacje o bezpieczeństwie dla bieżącego zadania za pomocą ścieżek nazw z obsługą narodowych wersji językowych.

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

- Wywołanie funkcji API socket() w celu uzyskania deskryptora gniazda.
- Wywołanie funkcji API SSL_Init() lub SSL_Init_Application() w celu zainicjowania środowiska pracy dla przetwarzania SSL i w celu określenia informacji o bezpieczeństwie SSL dla wszystkich sesji SSL, które będą uruchamiane w bieżącym zadaniu. Należy użyć tylko jednej z wymienionych funkcji API. Zalecane jest użycie funkcji API SSL_Init_Application().
- Wywołania gniazd w celu uaktywnienia połączenia. Aplikacja wywołuje funkcję API connect() w celu uaktywnienia połączenia z programem klienckim lub funkcje API bind(), listen() i accept() w celu umożliwienia serwerowi akceptowania przychodzących żądań połączeń.
- Wywołanie funkcji API SSL_Create() w celu włączenia obsługi SSL dla podłączonego gniazda.
- Wywołanie funkcji API SSL_Handshake() w celu zainicjowania negocjacji parametrów szyfrowania podczas uzgadniania SSL.

Uwaga: Zwykle aby uzgadnianie SSL się powiodło, program serwera musi okazać certyfikat. Serwer musi ponadto mieć dostęp do klucza prywatnego powiązanego z certyfikatem serwera i do zbioru bazy danych kluczy, w którym jest przechowywany certyfikat. W niektórych przypadkach także klient musi okazać certyfikat podczas przetwarzania uzgadniania SSL. Dotyczy to sytuacji, gdy na serwerze, z którym łączy się klient, włączono uwierzytelnianie klienta. Funkcje API SSL_Init() i SSL_Init_Application() identyfikują (choć w różny sposób) zbiór bazy danych kluczy, z którego są pobierane certyfikat i klucz prywatny, używane podczas uzgadniania.

- Wywołania funkcji API SSL_Read() i SSL_Write() w celu odebrania i wysłania danych.
- Wywołanie funkcji API SSL_Destroy() w celu wyłączenia obsługi SSL dla gniazda.
- Wywołanie funkcji API close() w celu usunięcia podłączonych gniazd.

Odsyłacze pokrewne

"Przykład: ustanawianie chronionego serwera za pomocą funkcji API SSL_" na stronie 145
Do tworzenia aplikacji chronionych można używać nie tylko funkcji API GSKit, lecz także funkcji API SSL_. Funkcje API SSL_ występują tylko w systemie operacyjnym i5/OS.

"Przykład: ustanawianie chronionego klienta za pomocą funkcji API SSL_" na stronie 151
Przykład ten zawiera aplikację klienta używającą funkcji API SSL_ do komunikowania się z aplikacją serwera używającą funkcji API SSL_.

Informacje pokrewne

Funkcja API socket() - tworzenie gniazd

Funkcja API listen() - nasłuchiwanie przychodzących żądań połączenia

Funkcja API bind() - ustawianie lokalnego adresu gniazda

Funkcja API connect() - nawiązywanie połączenia lub ustanawianie adresu docelowego

Funkcja API accept() - oczekiwanie na żądanie i nawiązywanie połączenia

Funkcja API close() - zamykanie deskryptora gniazda lub pliku

Funkcja API SSL_Create() - włączanie obsługi środowiska SSL dla podanego deskryptora gniazda

Funkcja API SSL_Destroy() - kończenie obsługi środowiska SSL dla podanej sesji SSL

Funkcja API SSL_Handshake() - inicjowanie protokołu uzgadniania środowiska SSL

Funkcja API SSL_Init() - inicjowanie bieżącego zadania dla środowiska SSL

Funkcja API SSL_Init_Application() - inicjowanie bieżącego zadania dla przetwarzania SSL na podstawie identyfikatora aplikacji

Funkcja API SSL_Read() - odbieranie danych z deskryptora gniazda z włączoną obsługą SSL

Funkcja API SSL_Write() - zapisywanie danych do deskryptora gniazda z włączoną obsługą SSL

Funkcja API SSL_Strerror() - odczytywanie komunikatu o błędzie wykonania w środowisku SSL

Funkcja API SSL_Perror() - drukowanie komunikatu o błędzie w środowisku SSL

Komunikaty z kodami błędów funkcji API gniazd chronionych

Aby odczytać komunikaty zawierające kody błędów funkcji API związanych z obsługą gniazd chronionych, należy wykonać następujące czynności.

1. W wierszu komend wpisz: DSPMSGD RANGE(*XXXXXXX*), gdzie *XXXXXXX* to identyfikator komunikatu dla danego kodu powrotu. Jeśli na przykład kod powrotu to 3, wpisz DSPMSGD RANGE(CPDBCB9).
2. Wybierz **1**, aby wyświetlić tekst komunikatu.

Tabela 14. Komunikaty z kodami błędów funkcji API gniazd chronionych

Kod powrotu	ID komunikatu	Nazwa stałej
0	CPCBC80	GSK_OK
1	CPDBCA1	GSK_INVALID_HANDLE
2	CPDBCB3	GSK_API_NOT_AVAILABLE
3	CPDBCB9	GSK_INTERNAL_ERROR

Tabela 14. Komunikaty z kodami błędów funkcji API gniazd chronionych (kontynuacja)

Kod powrotu	ID komunikatu	Nazwa stałej
4	CPC3460	GSK_INSUFFICIENT_STORAGE
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
502	CPE3406	GSK_WOULD_BLOCK
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
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

Tabela 14. Komunikaty z kodami błędów funkcji API gniazd chronionych (kontynuacja)

Kod powrotu	ID komunikatu	Nazwa stałej
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
-1011	CPDBCCB	SSL_ERROR_NO_CERTIFICATE_AUTHORITIES
-9998	CPDBCD8	SSL_ERROR_NO_REUSE

Odsyłacze pokrewne

“Przykłady: nawiązywanie połączeń chronionych” na stronie 119
Do utworzenia chronionego serwera i klienta można użyć funkcji API GSKit lub funkcji SSL_.

Obsługa klienta mechanizmu SOCKS

System operacyjny i5/OS obsługuje mechanizm SOCKS wersja 4. Pozwala to programom używającym rodziny adresów AF_INET z gniazdami typu SOCK_STREAM komunikować się z programami serwerowymi działającymi w systemach zlokalizowanych poza zaporą firewall.

Firewall to bardzo bezpieczny host, umieszczany między chronioną siecią wewnętrzną a niechronioną siecią zewnętrzną. Zwykle taka konfiguracja sieci nie umożliwia komunikacji między chronionym hostem a niechronioną siecią. Serwery proxy znajdujące się na firewallu pomagają zarządzać niezbędną komunikacją między chronionymi hostami a sieciami niechronionymi.

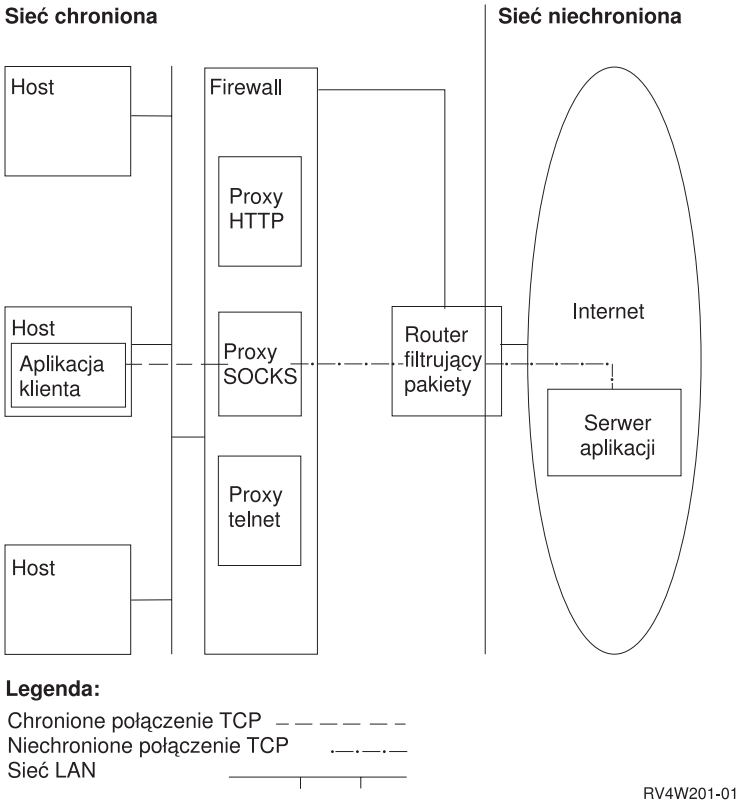
Aplikacje działające na serwerze w chronionej sieci wewnętrznej muszą wysyłać swoje żądania przez serwery proxy zapory firewall. Serwery proxy mogą następnie przekazać żądania do serwera istniejącego w sieci niechronionej. Dodatkowo mogą przekazać odpowiedź do aplikacji działającej na hoście rozpoczynającym komunikację. Typowym przykładem serwera proxy jest serwer proxy HTTP. Serwery proxy wykonują następujące zadania dla klientów HTTP:

- ukrywają sieć wewnętrzną przed systemami zewnętrznymi,
- 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 zaporze firewall jest uruchomienie bardziej wydajnego serwera proxy, znanego jako serwer SOCKS. Serwer SOCKS może działać jako serwer proxy dla dowolnego połączenia klienta TCP nawiązanego za pomocą funkcji API gniazd. Główną zaletą obsługi klienta SOCKS w systemie i5/OS jest umożliwienie aplikacjom klienta przezroczystego dostępu do serwera SOCKS bez konieczności modyfikowania kodu klienta.

Na poniższym rysunku przedstawiono typowy projekt zapory firewall z serwerami proxy HTTP, proxy telnet i proxy SOCKS. Należy zaznaczyć, że do ochrony dostępu klienta do serwera w Internecie są używane dwa oddzielne połączenia TCP. Jedno z nich 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. Skonfiguruj serwer SOCKS.
2. W chronionym systemie klienckim zdefiniuj wszystkie wychodzące połączenia TCP klienta, które mają być kierowane do serwera SOCKS w tym systemie klienckim.

   Aby skonfigurować obsługę klienta SOCKS, wykonaj poniższe czynności:

   a. W programie System i Navigator rozwiń gałąź ***system*** **→ Sieć → Konfiguracja TCP/IP** (system > Network > TCP/IP Configuration).
   b. Prawym klawiszem myszy kliknij opcję **Konfiguracja TCP/IP**.
   c. Kliknij **Właściwości**.
   d. Kliknij zakładkę **SOCKS**.
   e. Na stronie SOCKS wprowadź informacje o połączeniu.

   Uwaga: Dane o konfiguracji chronionego klienta SOCKS zostaną zapisane w zbiorze QASOSCFG w bibliotece QUSRSYS w systemie hosta chronionego klienta.

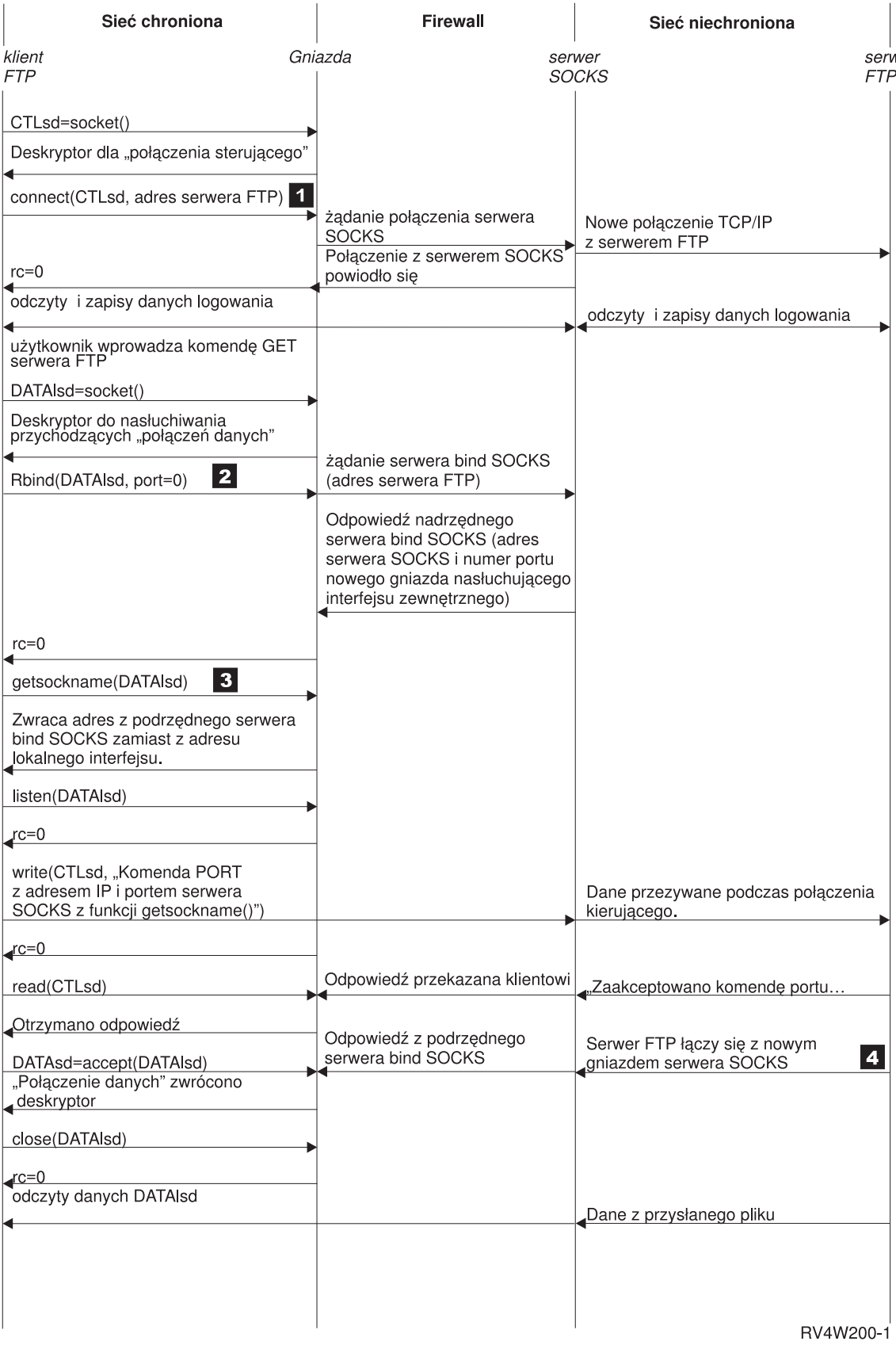
Po skonfigurowaniu system automatycznie przekieruje ustalone połączenia wychodzące do serwera SOCKS podanego w stronie SOCKS. Do aplikacji chronionego klienta nie trzeba wprowadzać żadnych zmian. Po odebraniu żądania serwer SOCKS ustanowi oddzielne połączenie zewnętrzne TCP/IP z serwerem w sieci niechronionej. Następnie serwer SOCKS przekaże dane między wewnętrznym i zewnętrznym połączeniem TCP/IP.

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

Dotychczas były omawiane 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 której zostanie utworzone połączenie

wychodzące przez serwer SOCKS. Połączenie przychodzące Rbind() musi pochodzić z tego samego adresu IP, który był użyty w połączeniu wychodzącym nawiązanym przez funkcję connect().

Na poniższym rysunku przedstawiono szczegółowy przegląd interakcji funkcji API gniazd z serwerem SOCKS w sposób przezroczysty dla aplikacji. W tym przykładzie klient FTP używa funkcji API Rbind() zamiast funkcji bind(), ponieważ protokół FTP pozwala serwerowi FTP na nawiązywanie połączeń do przesyłania danych w odpowiedzi na żądanie wysłania plików lub danych z klienta FTP. Funkcja jest wywoływana dzięki ponownej kompilacji kodu klienta FTP z definicją preprocesora (#define) __Rbind, która definiuje funkcję bind() jako Rbind(). Alternatywnie aplikacja może mieć jawnie zakodowaną funkcję Rbind() w odpowiednim kodzie źródłowym. Jeśli aplikacja nie wymaga połączeń przychodzących poprzez serwer SOCKS, nie należy używać funkcji Rbind().


Sieć chroniona
Firewall
Sieć niechroniona
klient FTP
Gniazda
serwer SOCKS
serwer FTP
CTLsd=socket()
Deskryptor dla „połączenia sterującego”
connect(CTLsd, adres serwera FTP) 1
żądanie połączenia serwera SOCKS
Nowe połączenie TCP/IP z serwerem FTP
Połączenie z serwerem SOCKS powiodło się
rc=0
odczyty i zapisy danych logowania
odczyty i zapisy danych logowania
użytkownik wprowadza komendę GET serwera FTP
DATAlsd=socket()
Deskryptor do nasłuchiwania przychodzących „połączeń danych”
żądanie serwera bind SOCKS (adres serwera FTP)
Rbind(DATAlsd, port=0) 2
Odpowiedź nadrzędnego serwera bind SOCKS (adres serwera SOCKS i numer portu nowego gniazda nasłuchującego interfejsu zewnętrznego)
rc=0
getsockname(DATAlsd) 3
Zwraca adres z podrzędnego serwera bind SOCKS zamiast z adresu lokalnego interfejsu.
listen(DATAlsd)
rc=0
write(CTLsd, „Komenda PORT z adresem IP i portem serwera SOCKS z funkcji getsockname()”)
Dane przezywane podczas połączenia kierującego.
rc=0
read(CTLsd)
Odpowiedź przekazana klientowi
„Zaakceptowano komendę portu…
Otrzymano odpowiedź
Odpowiedź z podrzędnego serwera bind SOCKS
Serwer FTP łączy się z nowym gniazdem serwera SOCKS 4
DATAsd=accept(DATAlsd)
„Połączenie danych” zwrócono deskryptor
close(DATAlsd)
rc=0
odczyty danych DATAlsd
Dane z przysłanego pliku
RV4W200-1

Uwagi:

1. Klient FTP inicjuje połączenie wychodzące TCP z niechronioną siecią poprzez serwer SOCKS. Adres docelowy podany przez klienta FTP podczas wywołania funkcji connect to adres IP i port serwera FTP znajdującego się w sieci niezabepieczonej. System chronionego hosta jest konfigurowany na stronie SOCKS tak, aby kierował to połączenie przez serwer SOCKS. Po skonfigurowaniu system automatycznie przekieruje połączenia do serwera SOCKS podanego na karcie SOCKS.
2. W celu nawiązania przychodzącego połączenia TCP gniazdo jest otwierane, po czym jest wywoływana funkcja Rbind(). Połączenie to 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 funkcji Rbind() muszą być nawiązywane bezpośrednio po połączeniu wychodzącym przez serwer SOCKS. Przed wywołaniem funkcji Rbind() niemożliwe jest przerwanie połączeń nieprzechodzących przez serwer SOCKS, odwołujących się do określonego wątku.
3. Funkcja getsockname() zwraca adres serwera SOCKS. Gniazdo jest logicznie powiązane z adresem IP serwera SOCKS połączonym z portem wybranym przez ten serwer. W przykładzie adres jest wysyłany przez ″połączenie sterujące″ Socket CTLsd do serwera FTP umieszczonego w sieci niechronionej. Jest to adres, z którym łączy się serwer FTP. Serwer FTP łączy się z serwerem SOCKS, a nie bezpośrednio z chronionym hostem.
4. Serwer SOCKS ustanawia połączenie dla danych z klientem FTP i przekazuje je między klientem a serwerem FTP. Wiele serwerów SOCKS zezwala na połączenie z chronionym klientem w ustalonym czasie. Jeśli w tym przedziale czasu do połączenia nie dojdzie, dla funkcji accept() wystąpi błąd ECONNABORTED.

Informacje pokrewne

Funkcja API bind() - ustawianie lokalnego adresu gniazda

Funkcja API connect() - nawiązywanie połączenia lub ustanawianie adresu docelowego

Funkcja API accept() - oczekiwanie na żądanie i nawiązywanie połączenia

Funkcja API getsockname() - wczytywanie lokalnego adresu gniazda

Funkcja API Rbind() - ustawianie zdalnego adresu gniazda

Ochrona wątków

Funkcja jest wątkowo bezpieczna (zapewnia ochronę wątków), jeśli w ramach tego samego procesu można ją uruchomić jednocześnie w wielu wątkach. Funkcję uznaje się za wątkowo bezpieczną tylko wtedy, gdy wszystkie funkcje przez nią wywoływane są również wątkowo bezpieczne. Funkcje API gniazd obejmują wątkowo bezpieczne funkcje systemowe i funkcje sieciowe.

Wszystkie funkcje sieciowe, których nazwy kończą się przyrostkiem ″_r″, mają podobną semantykę i są również wątkowo bezpieczne.

Inne procedury tłumaczące realizują wzajemnie ochronę wątków, ale korzystają ze struktury danych _res. Ta struktura danych jest współużytkowana między wszystkimi wątkami w procesie i może być zmieniana przez aplikację podczas wywołania procedury tłumaczącej.

Odsyłacze pokrewne

“Przykład: używanie funkcji gethostbyaddr_r() dla wątkowo bezpiecznych procedur sieciowych” na stronie 153
Przedstawiony program przykładowy używa funkcji API gethostbyaddr_r(). Wszystkie pozostałe procedury, których nazwy kończą się przyrostkiem *_r*, mają podobną semantykę i są również wątkowo bezpieczne.

“Przykład: aktualizacja systemu DNS i wysyłanie do niego zapytań” na stronie 177
W tym przykładzie przedstawiono sposób wysyłania zapytań do systemu nazw domen (DNS) oraz aktualizowania jego rekordów.

Nieblokujące operacje we/wy

Kiedy aplikacja wywoła jedną z wejściowych funkcji API gniazd, a nie będzie żadnych danych do odczytu, to funkcja API się zablokuje i nie zakończy działania aż do chwili pojawienia się danych do odczytu.

Podobnie aplikacja może zablokować wyjściową funkcję API gniazd, gdy ta nie będzie mogła natychmiast wysłać danych. Funkcje connect() oraz accept() mogą się ponadto zablokować podczas oczekiwania na nawiązanie połączenia z innymi programami.

Gniazda udostępniają aplikacjom metody umożliwiające wywoływanie funkcji API podlegających blokowaniu w taki sposób, że mogą one zostać zakończone bez opóźnienia. Metody te polegają na wywołaniu funkcji fcntl(), która włącza opcję **O_NONBLOCK**, lub funkcji ioctl() włączającej opcję **FIONBIO**. Po uruchomieniu trybu nieblokującego, jeśli funkcja API nie może być zakończona bez zablokowania, zostaje zakończona natychmiast. Funkcja connect() może zwrócić wartość [EINPROGRESS], który oznacza, że zostało uruchomione inicjowanie połączenia. Następnie można wybrać funkcję poll() lub select() w celu określenia, kiedy połączenie zostało zakończone. W przypadku wszystkich innych funkcji API, na które może mieć wpływ praca w trybie nieblokującym, kod błędu [EWOULDBLOCK] wskazuje, że wywołanie się nie powiodło.

Trybu nieblokującego można użyć z następującymi funkcjami API 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()

Odsyłacze pokrewne

“Przykład: nieblokujące operacje we/wy i funkcja select()” na stronie 155
W tym przykładzie przedstawiono aplikację serwera, która wykorzystuje nieblokujące operacje we/wy i funkcję API select().

Informacje pokrewne

Funkcja API fcntl() - wykonywanie komend sterujących plikami

Funkcja API accept() - oczekiwanie na żądanie i nawiązywanie połączenia

Funkcja API ioctl() - wykonywanie żądań sterowania operacjami we/wy

Funkcja API recv() - odbieranie danych

Funkcja API send() - wysyłanie danych

Funkcja API connect() - nawiązywanie połączenia lub ustanawianie adresu docelowego

Funkcja API gsk_secure_soc_read() - odbieranie danych podczas sesji chronionej

Funkcja API gsk_secure_soc_write() - wysyłanie danych podczas sesji chronionej

Funkcja API SSL_Read() - odbieranie danych z deskryptora gniazda z włączoną obsługą SSL

Funkcja API SSL_Write() - zapisywanie danych do deskryptora gniazda z włączoną obsługą SSL

Funkcja API read() - odczytywanie danych z deskryptora

Funkcja API readv() - odczytywanie danych z deskryptora przy użyciu wielu buforów

Funkcja API recvfrom() - odbieranie danych

Funkcja API recvmsg() - odbieranie komunikatów za pośrednictwem gniazda

Funkcja API send_file() - wysyłanie pliku za pomocą połączenia przez gniazdo

Funkcja API send_file64()

Funkcja API sendmsg() - wysyłanie komunikatów przez gniazdo

Funkcja API sendto() - wysyłanie danych

Funkcja API write() - zapisywanie danych do deskryptora

Funkcja API writev() - zapisywanie danych do deskryptora przy użyciu wielu buforów

Sygnały

Aplikacja może zażądać asynchronicznego powiadomienia (żądanie wysłania przez system *sygnału*) o wystąpieniu warunku, na który czeka.

Istnieją dwa sygnały asynchroniczne, które są wysyłane do aplikacji przez gniazda.

1. Sygnał *SIGUR*G jest wysyłany wtedy, gdy gniazdo obsługujące dane OOB odbierze tego rodzaju dane. Na przykład gniazdo typu SOCK_STREAM z rodziny adresów AF_INET można skonfigurować tak, aby wysyłało sygnał SIGURG.
2. Sygnał *SIGIO* jest wysyłany wtedy, gdy gniazdo dowolnego typu odbierze normalne dane lub dane OOB, gdy wystąpią warunki błędów lub gdy zdarzy się cokolwiek innego.

Przed zażądaniem od systemu wysyłania sygnałów aplikacja powinna sprawdzić, czy jest w stanie odebrać sygnał. Można tego dokonać poprzez skonfigurowanie procedury *obsługi sygnału*. Jednym ze sposobów ustawienia tej procedury jest wywołanie funkcji sigaction().

Aplikacja żąda od systemu wysłania sygnału *SIGURG* poprzez:

- wywołanie funkcji fcntl() oraz określenie identyfikatora procesu lub grupy procesów komendy F_SETOWN.
- wywołanie funkcji ioctl() oraz określenie komendy (żądania) FIOSETOWN lub SIOCSPGRP.

Aplikacja żąda od systemu wysłania sygnału *SIGIO* w dwóch etapach. W pierwszym, jak opisano powyżej, należy ustawić dla sygnału *SIGURG* identyfikator procesu lub grupy procesów. System uzyskuje dzięki temu informację, gdzie ma być dostarczony sygnał. W etapie drugim aplikacja musi wykonać jedną z następujących czynności:

- wywołanie funkcji fcntl() i określenie komendy F_SETFL z opcją FASYNC.
- wywołanie funkcji ioctl() i określenie komendy FIOASYNC.

Zmusza to system do wygenerowania sygnału SIGIO. Należy zaznaczyć, że powyższe czynności można wykonać w dowolnej kolejności. Jeśli ponadto aplikacja wysyła te żądania w gnieździe nasłuchującym, ustawiane wartości są dziedziczone przez wszystkie gniazda i zwracane do aplikacji za pomocą funkcji API accept(). Dzięki temu nowo zaakceptowane gniazda będą miały te same identyfikatory procesu lub grupy procesów, a także te same informacje w odniesieniu do wysyłanego sygnału SIGIO.

Gniazdo może generować sygnały asynchroniczne także w przypadku wystąpienia warunków błędu. Kiedy aplikacja odbierze [EPIPE] jako wartość *errno* dla funkcji API gniazda, wysyła sygnał SIGPIPE do tego procesu, który wywołał operację zakończoną wartością *errno*. W implementacji BSD sygnał SIGPIPE domyślnie kończy proces, który zakończył się wartością *errno*. Aby zachować zgodność z poprzednimi wersjami systemui5/OS, implementacja i5/OS domyślnie ignoruje sygnał SIGPIPE. Dzięki temu dodanie funkcji API obsługującej sygnały nie będzie miało negatywnego wpływu na istniejące aplikacje.

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

- accept()

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

Istotne jest to, że sygnały nie udostępniają aplikacjom deskryptorów gniazd identyfikujących miejsce wystąpienia sygnalizowanego warunku. Jeśli aplikacja używa wielu deskryptorów gniazd, musi albo wysyłać zapytania do poszczególnych deskryptorów, albo wywołać funkcję select() w celu określenia przyczyny odebrania sygnału.

Pojęcia pokrewne

“Dane spoza pasma” na stronie 62
Dane spoza pasma (out-of-band - OOB) są danymi specyficznymi dla użytkownika, które mają znaczenie tylko dla gniazd zorientowanych na połączenie (strumieniowych).

Odsyłacze pokrewne

“Przykład: używanie sygnałów z blokującymi funkcjami API gniazd” na stronie 168
Sygnały mogą przekazywać powiadomienie o tym, że doszło do zablokowania procesu lub aplikacji. Udostępniają także limit czasu blokowania procesów.

Informacje pokrewne

Funkcja API accept() - oczekiwanie na żądanie i nawiązywanie połączenia

Funkcja API sendmsg() - wysyłanie komunikatów przez gniazdo

Funkcja API sendto() - wysyłanie danych

Funkcja API write() - zapisywanie danych do deskryptora

Funkcja API writev() - zapisywanie danych do deskryptora przy użyciu wielu buforów

Funkcja API read() - odczytywanie danych z deskryptora

Funkcja API readv() - odczytywanie danych z deskryptora przy użyciu wielu buforów

Funkcja API connect() - nawiązywanie połączenia lub ustanawianie adresu docelowego

Funkcja API recvfrom() - odbieranie danych

Funkcja API recvmsg() - odbieranie komunikatów za pośrednictwem gniazda

Funkcja API recv() - odbieranie danych

Funkcja API send() - wysyłanie danych

Funkcja API select() - oczekiwanie na wydarzenia w wielu gniazdach

Rozsyłanie grupowe IP

Rozsyłanie grupowe IP pozwala aplikacjom na wysłanie pojedynczego datagramu IP, który zostanie odebrany przez grupę hostów w sieci.

Hosty należące do grupy mogą się znajdować w tej samej podsieci lub w różnych podsieciach połączonych routerami obsługującymi rozsyłanie grupowe. Hosty mogą dołączać do grup i opuszczać je w każdej chwili. Nie ma ograniczeń dotyczących położenia ani liczby elementów grupy hostów. Grupę hostów AF_INET identyfikuje klasa D adresów IP w zakresie od 224.0.0.1 do 239.255.255.255. Dla adresów rodziny AF_INET6 adres IPv6 zaczynający się od FF00::/8

jest rozpoznawany jako adres rozsyłania grupowego. Więcej informacji na ten temat można znaleźć w dokumencie RFC 3513: ″Internet Protocol Version 6 (IPv6) Addressing Architecture″ .

Obecnie można używać rozsyłania grupowego IP dla rodzin AF_INET i AF_INET6.

Aplikacja może wysyłać i odbierać datagramy rozsyłania grupowego za pomocą funkcji API gniazd oraz 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 API setsockopt() do sterowania parametrem rozsyłania grupowego przypisanym do tego gniazda. Funkcja API setsockopt() akceptuje następujące opcje poziomów IPPROTO_IP:

- IP_ADD_MEMBERSHIP: dołącza do podanej grupy rozsyłania.
- IP_DROP_MEMBERSHIP: opuszcza podaną grupę rozsyłania.
- IP_MULTICAST_IF: ustawia interfejs, poprzez który są wysyłane wychodzące datagramy rozsyłania grupowego.
- IP_MULTICAST_TTL: ustawia wartość Time To Live (TTL) w nagłówku IP wychodzących datagramów rozsyłania grupowego.
- IP_MULTICAST_LOOP: określa, czy kopia wychodzącego datagramu rozsyłania grupowego ma być dostarczona do hosta wysyłającego, o ile należy on do grupy rozsyłania.

Funkcja API setsockopt() akceptuje również następujące opcje poziomów IPPROTO_IPv6:

- IPv6_MULTICAST_IF: konfiguruje interfejs, przez który są wysyłane wychodzące datagramy rozsyłania grupowego.
- IPv6_MULTICAST HOPS: określa wartości limitu przeskoku używane do następnych pakietów rozsyłania grupowego wysyłanych przez gniazdo.
- IPv6_MULTICAST_LOOP: określa, czy kopia wychodzącego datagramu rozsyłania ma być dostarczona do hosta wysyłającego, o ile należy on grupy rozsyłania.
- IPv6 JOIN GROUP: dołącza do podanej grupy rozsyłania grupowego.
- IPv6 LEAVE GROUP: opuszcza podaną grupę rozsyłania grupowego.

Odsyłacze pokrewne

“Przykłady: rozsyłanie grupowe za pomocą rodziny adresów AF_INET” na stronie 172
Rozsyłanie grupowe IP pozwala aplikacjom na wysłanie pojedynczego datagramu IP, który zostanie odebrany przez grupę hostów w sieci.

Informacje pokrewne

Funkcja API setsockopt_ioctl() - ustawianie opcji gniazd

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

Obsługa gniazd w systemie i5/OS obejmuje funkcje API send_file() i accept_and_recv() umożliwiające szybsze i łatwiejsze przesyłanie danych między połączonymi gniazdami.

Te dwie funkcje API są szczególnie użyteczne dla aplikacji przesyłających pliki, takich jak serwery HTTP.

Pojedyncze wywołanie funkcji API send_file() pozwala na przesyłanie danych zbioru bezpośrednio z systemu plików poprzez połączone gniazdo.

Funkcja accept_and_recv() jest kombinacją trzech funkcji API gniazd: accept(), getsockname() oraz recv().

Odsyłacze pokrewne

“Przykład: przesyłanie danych za pomocą funkcji API send_file() i accept_and_recv()” na stronie 181
Poniższe programy przykładowe pozwalają serwerowi na komunikowanie się z klientem za pomocą funkcji API send_file() i accept_and_recv().

Informacje pokrewne

Funkcja API send_file() - wysyłanie pliku za pomocą połączenia przez gniazdo

Funkcja API 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 następuje powiadomienie programu B o ich nadejściu.

Dane OOB są obsługiwane tylko przez rodziny adresów AF_INET (SOCK_STREAM) i AF_INET6 (SOCK_STREAM).

Dane OOB są wysyłane poprzez ustawienie opcji MSG_OOB w funkcjach API send(), sendto() i sendmsg().

Transmisja danych OOB jest taka sama jak zwykłych danych. Są one wysyłane po danych buforowanych. Innymi słowy, dane OOB nie mają pierwszeństwa przed danymi buforowanymi; 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. Wskazuje on na ostatni bajt w wysłanych danych OOB.

  Uwaga: Wartość określająca, który na bajt wskazuje znacznik OOB, jest ustawiana globalnie dla całego systemu (korzystają z niej wszystkie aplikacje). Wartość ta powinna być spójna między lokalnym i zdalnym końcem połączenia TCP. Musi też być używana zgodnie przez aplikacje typu klient i typu serwer.

  Żądanie SIOCATMARK ioctl() określa, czy wskaźnik odczytu wskazuje na ostatni bajt danych OOB.

  Uwaga: Jeśli wysłano wiele pakietów danych OOB, znacznik OOB będzie wskazywał na ostatni bajt końcowego ich wystąpienia.
- Jeśli dane OOB zostały wysłane, operacja wejściowa będzie przetwarzała dane do znacznika OOB, niezależnie od tego, czy dane OOB zostały odebrane.
- Do odbioru danych OOB są używane funkcje API recv(), recvmsg() lub recvfrom() (z ustawioną opcją MSG OOB). Po zakończeniu realizacji funkcji API, jeśli zdarzy się jeden z przypadków wymienionych poniżej, zwracany jest błąd [EINVAL]:
  - nie została ustawiona opcja gniazda SO_OOBINLINE 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 dane normalne. (Dane normalne to te, które program może odebrać bez ustawiania opcji MSG_OOB). Ostatni bajt z wysłanych danych OOB nie jest przechowywany w strumieniu danych normalnych. Bajt ten można odebrać jedynie poprzez wywołanie funkcji API recv(), recvmsg() lub recvfrom() (z ustawioną opcją MSG_OOB). Jeśli odbiór danych odbywa się bez ustawionej opcji MSG_OOB i jeśli są odbierane dane normalne, bajt OOB jest usuwany. Jeśli ponadto wysłano wiele pakietów danych OOB, wcześniejsze dane są niszczone, zapamiętywane jest natomiast tylko ostatnie wystąpienie danych OOB.

  Jeśli opcja gniazda SO_OOBINLINE została ustawiona, wszystkie wysyłane dane są przechowywane w strumieniu danych normalnych. Dane można odtworzyć przez wywołanie jednej z trzech funkcji API odbierania bez ustawiania opcji MSG_OOB (jeśli zostanie ona określona, zostanie zwrócony błąd [EINVAL]). Jeśli zostanie wysłanych wiele pakietów danych OOB, nie zostaną one utracone.
- Jeśli nie zostanie ustawiona opcja SO_OOBINLINE, dane OOB nie są odrzucane; zostają one odebrane, następnie użytkownik ustawia opcję SO_OOBINLINE. Początkowy bajt OOB jest uznawany za normalne dane.

- Jeśli opcja SO_OOBINLINE nie została ustawiona, dane OOB zostały wysłane, a program odbierający wywołał funkcję API wejścia w celu ich odebrania, znacznik OOB będzie w dalszym ciągu poprawny. Program odbierający może nadal sprawdzić, czy odczytana wartość wskazuje na znacznik OOB, nawet jeśli bajt OOB został odczytany.

  Pojęcia pokrewne

  "Sygnały" na stronie 59
  Aplikacja może zażądać asynchronicznego powiadomienia (żądanie wysłania przez system *sygnału*) o wystąpieniu warunku, na który czeka.

  Informacje pokrewne

  Funkcja API sendmsg() - wysyłanie komunikatów przez gniazdo

  Komenda Zmiana atrybutów TCP/IP (Change TCP/IP Attributes - CHGTCPA)

Multipleksowanie we/wy - funkcja select()

Zalecane jest używanie asynchroniczne operacji we/wy, ponieważ zapewniają bardziej efektywną metodę maksymalizowania zasobów aplikacji niż funkcja API select(). Konkretny projekt aplikacji może jednak dopuszczać użycie funkcji API select().

Podobnie jak asynchroniczne operacje we/wy funkcja API select() tworzy wspólny punkt jednoczesnego oczekiwania na wiele warunków. Jednak funkcja select() umożliwia aplikacji określenie zestawu deskryptorów w celu sprawdzenia, czy:

- są dane do odczytu,
- dane mogą zostać zapisane,
- wystąpił wyjątek.

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

Funkcja API select() umożliwia również aplikacjom określenie, czy w czasie oczekiwania na dane chcą pozostać dostępne. Aplikacje mogą określić czas oczekiwania.

Odsyłacze pokrewne

"Przykład: nieblokujące operacje we/wy i funkcja select()" na stronie 155
W tym przykładzie przedstawiono aplikację serwera, która wykorzystuje nieblokujące operacje we/wy i funkcję API select().

Funkcje sieciowe gniazd

Funkcje sieciowe gniazd umożliwiają aplikacjom uzyskiwanie informacji od hostów, protokołów, usług oraz plików sieciowych.

Dostęp do tych informacji jest możliwy poprzez nazwę, adres lub poprzez dostęp sekwencyjny. Te funkcje (lub procedury) sieciowe są wymagane podczas konfigurowania komunikacji między programami działającymi w sieci i nie są używane przez gniazda rodziny AF_UNIX.

Procedury są następujące:

- 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 internetowych adresach sieciowych,
- przekształcają adresy IP i notacje dziesiętne z kropkami.

W skład procedur sieciowych wchodzą tak zwane 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 również być wywoływane bezpośrednio. Procedury tłumaczące są używane głównie do uzyskiwania dostępu do systemu nazw domen (DNS) poprzez aplikacje używające gniazd.

Pojęcia pokrewne

“Charakterystyka gniazd” na stronie 6
Gniazda mają następujące cechy wspólne.

Odsyłacze pokrewne

“Przykład: używanie funkcji gethostbyaddr_r() dla wątkowo bezpiecznych procedur sieciowych” na stronie 153
Przedstawiony program przykładowy używa funkcji API gethostbyaddr_r(). Wszystkie pozostałe procedury, których nazwy kończą się przyrostkiem *_r*, mają podobną semantykę i są również wątkowo bezpieczne.

“System nazw domen”
System operacyjny umożliwia aplikacjom dostęp do systemu nazw domen (DNS) za pośrednictwem funkcji tłumaczących.

Informacje pokrewne

Funkcje systemu gniazd

Funkcja API gethostbyname() - pobieranie informacji o hoście według nazwy hosta

Funkcja API getaddrinfo() - pobieranie informacji o adresie

Funkcja API gethostbyaddr() - pobieranie informacji o hoście według adresu IP

Funkcja API getnameinfo() - pobieranie informacji o nazwie według adresu gniazda

System nazw domen

System operacyjny umożliwia aplikacjom dostęp do systemu nazw domen (DNS) za pośrednictwem funkcji tłumaczących.

Na system DNS składają się trzy główne komponenty:

Rekordy zasobów i przestrzeni nazw domen
Specyfikacje przestrzeni nazw zorganizowanych w strukturę drzewa oraz danych związanych z nazwami.

Serwery nazw
Programy serwerowe, które przechowują informacje o strukturze drzewa domeny oraz ustawiają informacje.

Programy tłumaczące
Programy, które wyodrębniają informacje z serwerów nazw w odpowiedzi na żądania klientów.

Programy tłumaczące dostępne w implementacji i5/OS 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 kodu ASCII na EBCDIC i z kodu EBCDIC na ASCII. Opcjonalnie do bezpiecznej komunikacji z serwerem DNS programy tłumaczące używają sygnatur transakcyjnych (TSIG).

Więcej informacji o nazwach domen można znaleźć w następujących dokumentach RFC, lokalizowanych za pomocą wyszukiwarki RFC .

- RFC 1034: Domain names - concepts and facilities.
- RFC 1035: Domain names - implementation and specification.
- RFC 1886: DNS Extensions to support IP version 6.
- RFC 2136: Dynamic Updates in the Domain Name System (DNS UPDATE).
- RFC 2181: Clarifications to the DNS Specification.
- RFC 2845: Secret Key Transaction Authentication for DNS (TSIG).
- RFC 3152: DNS Delegation of IP6.ARPA.

Odsyłacze pokrewne

“Funkcje sieciowe gniazd” na stronie 63
Funkcje sieciowe gniazd umożliwiają aplikacjom uzyskiwanie informacji od hostów, protokołów, usług oraz plików sieciowych.

Informacje pokrewne

System DNS

Funkcje systemu gniazd

Zmienne środowiskowe

Zmiennych środowiskowych można używać do wymuszania domyślnych parametrów początkowych funkcji tłumaczenia nazw.

Zmienne środowiskowe są sprawdzane tylko po pomyślnym wywołaniu funkcji res_init() lub res_ninit(). Jeśli więc struktura została zainicjowana ręcznie, zmienne środowiskowe są ignorowane. Należy ponadto zwrócić uwagę na to, że struktura jest inicjowana tylko raz, dlatego późniejsze modyfikacje zmiennych środowiskowych również będą ignorowane.

Uwaga: Nazwę zmiennej środowiskowej należy pisać wielkimi literami. Wartości łańcuchów mogą natomiast zawierać zarówno małe, jak i wielkie litery. Do zapisywania nazw zmiennych środowiskowych oraz wartości w systemach japońskich używających identyfikatora CCSID 290 należy używać wyłącznie wielkich liter i cyfr. Na poniższej liście zamieszczono opisy zmiennych środowiskowych, których można używać razem z funkcjami API res_init() i res_ninit().

LOCALDOMAIN

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

RES_OPTIONS

Zmienna środowiskowa RES_OPTIONS umożliwia modyfikację pewnych zmiennych wewnętrznych programu tłumaczącego. Zmienna ta może przyjmować jedną lub więcej wartości opisanych poniżej, oddzielonych spacjami.

- **NDOTS: n** Określa maksymalną liczbę kropek, jakie może zawierać nazwa przekazana do funkcji res_query(), zanim zostanie wykonane początkowe zapytanie absolutne. Wartością domyślną dla n jest 1, co oznacza, że jeśli w nazwie występuje co najmniej jedna kropka, to najpierw zostanie podjęta próba przetłumaczenia tej nazwy jako absolutnej, a w razie niepowodzenia będą do niej dołączane kolejne 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 zrezygnuje i zwróci się do następnego serwera nazw na liście.
- **ROTATE:** Ustawia wartość RES_ROTATE w strukturze danych _res.options, która powoduje rotację serwerów nazw na liście. Powoduje to równomierne rozłożenie między serwery obciążenia związanego z zapytaniami, co zapobiega sytuacji, w której wszystkie klienty 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 kodu ASCII czy znaki sterujące.

QIBM_BIND_RESOLVER_FLAGS

Ta zmienna środowiskowa zawiera listę opcji programu tłumaczącego oddzielonych spacjami. Spowoduje to zignorowanie opcji RES_DEFAULT (struktura state.options) i wartości skonfigurowanych w systemie za pomocą komendy 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 skonfigurowanych wartości komendy CHGTCPDMN. Następnie zostanie ona użyta do przesłonięcia wartości domyślnych. Opcje określone w tej zmiennej środowiskowej można poprzedzić symbolami '+', '-' lub 'NOT_', aby ustawić ('+') lub zresetować ('-','NOT_') wartość.

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

```
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 listę maksymalnie dziesięciu oddzielonych spacjami par adresów IP i masek w postaci liczb dziesiętnych oddzielonych kropkami (9.5.9.0/255.255.255.0), stanowiącą listę sortowania (struktura state.sort_list).

Informacje pokrewne

res_init()

res_ninit()

res_query()

Buforowanie danych

Celem buforowania odpowiedzi na zapytania serwera DNS wykonywanego przez gniazda systemu i5/OS jest zmniejszenie ruchu w sieci. Pamięć podręczna jest dodawana i aktualizowana, gdy jest to wymagane.

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

Jeśli odpowiedzi z sieci są autorytatywne, są one zapisywane w pamięci podręcznej. Odpowiedzi nieautorytatywne nie są buforowane. Buforowane nie są także odpowiedzi odebrane jako wynik zapytania odwrotnego. Zawartość tej pamięci podręcznej można usunąć poprzez zaktualizowanie konfiguracji serwera DNS za pomocą komend Zmiana domeny TCP/IP (Change TCP/IP Domain - CHGTCPDMN) lub Konfigurowanie TCP/IP (Configure TCP/IP - CFGTCP) albo za pomocą programu System i Navigator.

Odsyłacze pokrewne

"Przykład: aktualizacja systemu DNS i wysyłanie do niego zapytań" na stronie 177
W tym przykładzie przedstawiono sposób wysyłania zapytań do systemu nazw domen (DNS) oraz aktualizowania jego rekordów.

Zgodność z Berkeley Software Distribution (BSD)

Gniazda są interfejsem systemu BSD.

Semantyka, na przykład kody powrotu otrzymywane przez aplikację oraz argumenty dostępne w obsługiwanych funkcjach, należy do systemu BSD. Niektóre reguły semantyki BSD nie są jednak dostępne w implementacji i5/OS, dlatego do uruchomienia w systemie typowej aplikacji opartej na gniazdach BSD mogą być potrzebne zmiany.

Na poniższej liście zestawiono różnice między implementacjami w systemachi5/OS i BSD.

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

/etc/hosty, /etc/usługi, /etc/sieci oraz /etc/protokoły

Dla tych plików implementacja i5/OS dostarcza następujące bazy danych.

/etc/resolv.conf

Implementacja i5/OS wymaga, aby informacje te zostały skonfigurowane za pośrednictwem strony właściwości TCP/IP w programie System i Navigator. Aby uzyskać dostęp do strony właściwości TCP/IP, wykonaj następujące czynności:

1. W programie System i Navigator rozwiń gałąź ***system*** **→ Sieć → Konfiguracja TCP/IP** (system > Network > TCP/IP Configuration).
2. Prawym klawiszem myszy kliknij opcję **Konfiguracja TCP/IP**.
3. Kliknij **Właściwości**.

Funkcja 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 i5/OS nie obsługuje tego scenariusza (wywołanie funkcji bind() się nie powiedzie).

Funkcja close()

Implementacja i5/OS obsługuje licznik czasu zwłoki dla funkcji API close() z wyjątkiem gniazd AF_INET działających poprzez sieć SNA (architektura systemów sieciowych). Niektóre implementacje systemu BSD nie obsługują licznika czasu zwłoki dla funkcji API close().

Funkcja connect()

Wywołanie w systemie BSD funkcji connect() w odniesieniu do gniazda, które zostało już połączone z adresem i korzysta z usługi transportu bezpołączeniowego z użyciem niepoprawnego adresu lub adresu o niepoprawnej długości, powoduje rozłączenie gniazda. Implementacja i5/OS nie obsługuje tego scenariusza (wywołanie funkcji connect() się nie powiedzie, a gniazdo pozostanie połączone).

Gniazdo z transportem bezpołączeniowym, dla którego została wywołana funkcja connect(), można odłączyć poprzez przypisanie parametrowi address_length wartości zero i wywołanie kolejnej funkcji connect().

Funkcje accept(), getsockname(), getpeername(), recvfrom() i recvmsg()

W przypadku rodziny adresów AF_UNIX lub AF_UNIX_CCSID, gdy gniazdo nie zostanie znalezione, domyślna implementacja i5/OS może zwrócić długość adresu równą zeru oraz nieokreśloną strukturę adresu. Implementacje i5/OS BSD 4.4/UNIX i inne mogą zwrócić małą strukturę adresu zawierającą tylko określenie rodziny adresów.

Funkcja ioctl()

- W systemie BSD dla gniazd typu SOCK_DGRAM żądanie FIONREAD zwróci długość danych oraz długość adresu. W implementacji i5/OS żądanie FIONREAD zwraca tylko długość danych.
- Nie wszystkie żądania dostępne w większości implementacji BSD funkcji ioctl() są dostępne w implementacji i5/OS funkcji ioctl().

Funkcja listen()

W systemie BSD wywołanie funkcji listen() z parametrem backlog o wartości mniejszej od zera nie powoduje błędu. Dodatkowo w pewnych 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 systemu i5/OS zwraca błąd. Nadanie parametrowi backlog poprawnej wartości powoduje, że zostanie ona użyta jako ten parametr. Natomiast nadanie parametrowi backlog wartości większej niż {SOMAXCONN} spowoduje, że przyjmie on wartość domyślną ustawioną dla {SOMAXCONN}.

Dane OOB (out-of-band)

W implementacji i5/OS dane OOB nie są usuwane, jeśli opcja SO_OOBINLINE została włączona dopiero po odebraniu danych OOB (wcześniej była nieustawiona). Początkowy bajt OOB jest uznawany 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.

Funkcje res_xlate() i res_close()

Te funkcje API są wbudowane w procedury tłumaczące implementacji i5/OS. Funkcja API res_xlate() tłumaczy pakiety systemu nazw domen (DNS) z kodu EBCDIC na kod ASCII oraz z kodu ASCII na kod EBCDIC. Funkcja API res_close() służy do zamykania gniazda, które było używane przez funkcję API res_send() z ustawioną opcją RES_STAYOPEN. Ponadto funkcja API res_close() zeruje strukturę _res.

Funkcje sendmsg() i recvmsg()

Implementacja i5/OS funkcji sendmsg() i recvmsg() umożliwia korzystanie z wektorów we/wy {MSG_MAXIOVLEN}. Implementacja BSD dopuszcza maksymalnie {MSG_MAXIOVLEN - 1} wektorów we/wy.

Sygnały

Istnieje kilka różnic związanych z obsługą sygnałów:

- Implementacje BSD wywołują sygnał SIGIO za każdym razem, gdy jest otrzymywane potwierdzenie dla danych wysyłanych przez operację wyjściową. Implementacja gniazd w systemiei5/OS nie generuje sygnałów powiązanych z danymi wychodzącymi.
- W implementacjach BSD działaniem domyślnym dla sygnału SIGPIPE jest zakończenie procesu. Aby zachować zgodność z poprzednimi wersjami systemu i5/OS, implementacja i5/OS domyślnie ignoruje sygnał SIGPIPE.

Opcja SO_REUSEADDR

W systemach BSD wywołanie funkcji connect() dla gniazda typu SOCK_DGRAM z rodziny AF_INET spowoduje, że system zmieni adres, z którym jest powiązane gniazdo, na adres interfejsu użytego do połączenia się z adresem podanym w funkcji API connect(). Jeśli na przykład gniazdo typu SOCK_DGRAM zostanie powiązane z adresem INADDR_ANY, a następnie połączone z adresem a.b.c.d, to 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 jest powiązane gniazdo, to na przykład a.b.c.e, to podczas wywołania funkcji API getsockname zamiast adresu INADDR_ANY pojawi się adres a.b.c.e i konieczne będzie użycie opcji SO_REUSEADDR w celu powiązania z adresem a.b.c.e wszelkich innych gniazd dla tego samego numeru portu.

W przypadku implementacji i5/OS przebieg tego procesu będzie inny, to znaczy adres lokalny nie zostanie zmieniony z INADDR_ANY na a.b.c.e. Funkcja API getsockname() będzie zwracać adres INADDR_ANY także po zrealizowaniu połączenia.

Opcje SO_SNDBUF i SO_RCVBUF

Wartości przypisane opcjom SO_SNDBUF i SO_RCVBUF w systemie BSD zapewniają poziom sterowania wyższy niż w przypadku implementacji i5/OS. W implementacji i5/OS wartości te spełniają funkcję pomocniczą.

Pojęcia pokrewne

"Działanie gniazd" na stronie 3
Gniazd używa się powszechnie do obsługi interakcji między klientem a serwerem. W typowej konfiguracji systemu serwer jest umieszczony na jednym komputerze, a klienty na innych komputerach. Klienty łączą się z serwerem, wymieniają informacje, a następnie się odłączają.

Odsyłacze pokrewne

"Przykład: używanie sygnałów z blokującymi funkcjami API gniazd" na stronie 168
Sygnały mogą przekazywać powiadomienie o tym, że doszło do zablokowania procesu lub aplikacji. Udostępniają także limit czasu blokowania procesów.

Informacje pokrewne

Funkcja API accept() - oczekiwanie na żądanie i nawiązywanie połączenia

Funkcja API sendmsg() - wysyłanie komunikatów przez gniazdo

Funkcja API connect() - nawiązywanie połączenia lub ustanawianie adresu docelowego

Funkcja API recvfrom() - odbieranie danych

Funkcja API recvmsg() - odbieranie komunikatów za pośrednictwem gniazda

Funkcja API bind() - ustawianie lokalnego adresu gniazda

Funkcja API getsockname() - wczytywanie lokalnego adresu gniazda

Funkcja API socket() - tworzenie gniazd

Funkcja API listen() - nasłuchiwanie przychodzących żądań połączenia

Funkcja API ioctl() - wykonywanie żądań sterowania operacjami we/wy

Funkcja API getpeername() - wczytywanie docelowego adresu gniazda

Funkcja API close() - zamykanie deskryptora gniazda lub pliku

Zgodność ze standardem UNIX 98

Standard UNIX 98, opracowany przez konsorcjum programistów i dostawców oprogramowania Open Group, usprawnił współdziałanie w ramach systemu operacyjnego UNIX. Uwzględniona w nim została większość funkcji związanych z obsługą Internetu, dzięki którym system UNIX stał się znany.

Gniazda systemu i5/OS umożliwiają programistom tworzenie aplikacji używających gniazd, kompatybilnych ze środowiskiem operacyjnym UNIX 98. IBM obsługuje obecnie dwie wersje większości funkcji API gniazd. Podstawowe funkcje API gniazd w systemie i5/OS używają struktury i składni Berkeley Socket Distribution (BSD) 4.3. W drugiej wersji używa się składni i struktur zgodnych z BSD 4.4 i specyfikacją interfejsu programistycznego UNIX 98. Poprzez zdefiniowanie dla makra _XOPEN_SOURCE wartości 520 lub większej można wybrać interfejs zgodny ze standardem UNIX 98.

Różnice w strukturze adresów dla aplikacji zgodnych z UNIX 98

Po podaniu makra _XOPEN_OPEN można pisać aplikacje zgodne ze standardem UNIX 98, korzystające z tych samych rodzin adresów, które są używane w domyślnych implementacjach systemui5/OS. Należy jednak zaznaczyć, że w strukturze adresów **sockaddr** występują różnice. W poniższej tabeli porównano strukturę adresów **sockaddr** BSD 4.3 ze strukturą adresów zgodną ze standardem UNIX 98:

Tabela 15. Porównanie struktur adresów gniazd BSD 4.3 i UNIX 98/BSD 4.4

Struktura BSD 4.3	Struktura BSD 4.4 zgodna ze standardem 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	

Tabela 15. Porównanie struktur adresów gniazd BSD 4.3 i UNIX 98/BSD 4.4 (kontynuacja)

Struktura BSD 4.3	**Struktura BSD 4.4 zgodna ze standardem 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];`
	`};`

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 kompilowania aplikacji napisanych 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. W poniższej tabeli przedstawiono funkcje API, których to dotyczy. Podczas pisania aplikacji używających gniazd w języku opartym na C można w bezpośredni sposób określać wewnętrzne nazwy tych funkcji API. Aby uzyskać informacje i uwagi na temat użycia obydwu wersji wymienionych funkcji API należy skorzystać z odsyłaczy.

Tabela 16. Funkcja API i odpowiadająca jej nazwa zgodna ze standardem UNIX 98

Nazwa funkcji API	**Nazwa wewnętrzna**
accept()	qso_accept98()
accept_and_recv()	qso_accept_and_recv98()
bind()	qso_bind98()
connect()	qso_connect98()
endhostent()	qso_endhostent98()
endnetent()	qso_endnetent98()
endprotoent()	qso_endprotoent98()
endservent()	qso_endservent98()
getaddrinfo()	qso_getaddrinfo98()
gethostbyaddr()	qso_gethostbyaddr98()
gethostbyaddr_r()	qso_gethostbyaddr_r98()
gethostname()	qso_gethostname98()
gethostname_r()	qso_gethostname_r98()
gethostbyname()	qso_gethostbyname98()

Tabela 16. Funkcja API i odpowiadająca jej nazwa zgodna ze standardem UNIX 98 (kontynuacja)

Nazwa funkcji API	Nazwa wewnętrzna
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()
recvfrom()	qso_recvfrom98()
recvmsg()	qso_recvmsg98()
send()	qso_send98()
sendmsg()	qso_sendmsg98()
sendto()	qso_sendto98()
sethostent()	qso_sethostent98()
setnetent()	qso_setnetent98()
setprotoent()	qso_setprotoent98()
setservent()	qso_setprotoent98()
setsockopt()	qso_setsockopt98()
shutdown()	qso_shutdown98()
socket()	qso_socket98()
socketpair()	qso_socketpair98()

Pojęcia pokrewne

“Działanie gniazd” na stronie 3
Gniazd używa się powszechnie do obsługi interakcji między klientem a serwerem. W typowej konfiguracji systemu serwer jest umieszczony na jednym komputerze, a klienty na innych komputerach. Klienty łączą się z serwerem, wymieniają informacje, a następnie się odłączają.

Informacje pokrewne

Funkcja API accept() - oczekiwanie na żądanie i nawiązywanie połączenia
Funkcja API accept_and_recv()
Funkcja API connect() - nawiązywanie połączenia lub ustanawianie adresu docelowego
Funkcja API sendmsg() - wysyłanie komunikatów przez gniazdo
Funkcja API recvfrom() - odbieranie danych
Funkcja API recvmsg() - odbieranie komunikatów za pośrednictwem gniazda
Funkcja API Rbind() - ustawianie zdalnego adresu gniazda
Funkcja API recv() - odbieranie danych
Funkcja API bind() - ustawianie lokalnego adresu gniazda
Funkcja API getsockname() - wczytywanie lokalnego adresu gniazda
Funkcja API socket() - tworzenie gniazd
Funkcja API socketpair() - tworzenie pary gniazd
Funkcja API listen() - nasłuchiwanie przychodzących żądań połączenia
Funkcja API ioctl() - wykonywanie żądań sterowania operacjami we/wy
Funkcja API getpeername() - wczytywanie docelowego adresu gniazda
Funkcja API close() - zamykanie deskryptora gniazda lub pliku
endhostent()
endnetent()
endprotoent()
endservent()
Funkcja API gethostbyname() - pobieranie informacji o hoście według nazwy hosta
Funkcja API getaddrinfo() - pobieranie informacji o adresie
Funkcja API gethostbyaddr() - pobieranie informacji o hoście według adresu IP
Funkcja API getnameinfo() - pobieranie informacji o nazwie według adresu gniazda
gethostname()
gethostent()
getnetbyaddr()
getnetbyname()
getnetent()
getprotobyname()
getprotobynumber()
getprotoent()
getsockopt()
getservbyname()
getservbyport()
getservent()
inet_addr()
inet_1naof()
inet_makeaddr()
inet_netof()
inet_network()
Funkcja API send() - wysyłanie danych
Funkcja API sendto() - wysyłanie danych
sethostent()

setnetent()

setprotoent()

setservent()

Funkcja API setsockopt_ioctl() - ustawianie opcji gniazd

Przekazywanie deskryptorów między procesami - funkcje sendmsg() i recvmsg()

Przekazanie otwartego deskryptora między zadaniami pozwala jednemu procesowi (zwykle procesowi serwera) wykonać wszystkie działania niezbędne do uzyskania deskryptora, w tym otworzyć plik, nawiązać połączenie i oczekiwać na zakończenie realizacji funkcji API accept(). Pozwala to także innemu procesowi (zwykle procesowi roboczemu) obsłużyć wszystkie operacje przesyłania danych, gdy tylko deskryptor zostanie otwarty.

Zdolność przekazywania otwartego deskryptora między zadaniami może prowadzić do nowego sposobu projektowania aplikacji klient/serwer. Upraszcza to logikę zadań serwera i procesu roboczego. Konstrukcja taka umożliwia także łatwą obsługę różnych typów procesów roboczych. Serwer może łatwo sprawdzić, któremu typowi procesu roboczego powinien zostać przekazany deskryptor.

Gniazda udostępniają trzy zestawy funkcji API pozwalające na przekazywanie deskryptorów między zadaniami serwera:

- Funkcja spawn()

  Uwaga: Funkcja spawn() nie jest funkcją API gniazd. Jest ona dostarczana wraz z innymi funkcjami API systemu i5/OS związanymi z procesami.
- Funkcje givedescriptor() i takedescriptor()
- Funkcje sendmsg() i recvmsg()

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

W porównaniu z funkcjami spawn() oraz givedescriptor() i takedescriptor() funkcje sendmsg() i recvmsg() mają jednak więcej zalet:

Przenośność
: Funkcje API givedescriptor() oraz takedescriptor() to funkcje niestandardowe, unikalne dla systemu operacyjnego i5/OS. Jeśli wymagane jest zachowanie przenośności aplikacji między systemem operacyjnym i5/OS a systemem UNIX, to lepszym rozwiązaniem będzie użycie funkcji API sendmsg() i recvmsg().

Komunikacja informacji sterujących
: Często się zdarza, że po otrzymaniu deskryptora zadanie procesu roboczego potrzebuje dodatkowych informacji:

  - jaki to rodzaj deskryptora,
  - co należy z nim zrobić.

  Funkcje API sendmsg() i recvmsg() pozwalają na przesyłanie wraz z deskryptorem danych, które mogą być informacjami sterującymi; funkcje givedescriptor() i takedescriptor() nie dają takiej możliwości.

Wydajność
: W porównaniu z aplikacjami korzystającymi z funkcji givedescriptor() i takedescriptor() aplikacje korzystające z funkcji sendmsg() i recvmsg() działają wydajniej pod względem:

  - czasu, jaki upłynął,
  - stopnia wykorzystania jednostki centralnej,
  - skalowalności.

  Wzrost wydajności aplikacji zależy od zakresu przekazywania deskryptorów przez tę aplikację.

Pula zadań procesów roboczych

Pulę zadań procesów roboczych można ustawić tak, aby serwer mógł przekazać deskryptor i aby otrzymało go tylko jedno (aktywne) zadanie z puli. Aby uzyskać taki efekt, 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 identyfikator zadania procesu roboczego

Funkcja givedescriptor wymaga, aby zadanie serwera znało identyfikator zadania procesu roboczego. Zadanie procesu roboczego uzyskuje zwykle identyfikator zadania i przekazuje go do zadania serwera z kolejką danych. Funkcje sendmsg i recvmsg nie wymagają dodatkowych czynności przy tworzeniu tej kolejki danych i zarządzaniu nią.

Adaptacyjny program serwera

Gdy serwer jest zaprojektowany z użyciem funkcji givedescriptor i takedescriptor, to przekazywanie identyfikatorów zadań od zadań roboczych do serwera zwykle odbywa się za pośrednictwem kolejki danych. Serwer wywołuje następnie funkcje API socket(), bind(), listen() i accept(). Po zakończeniu działania funkcji accept() serwer przejdzie do następnego dostępnego identyfikatora zadania z kolejki danych. Następnie przekaże połączenie przychodzące do zadania procesu roboczego. Problemy pojawiają się wówczas, gdy jednocześnie wystąpi wiele żądań połączeń przychodzących i nie znajdzie się dla nich wystarczająco dużo 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 żadna z tych możliwości nie jest wskazana, gdyż dodatkowe żądania przychodzące mogą wypełnić kolejkę nasłuchiwania.

Serwery, które do przekazywania deskryptorów wykorzystują funkcje sendmsg() i recvmsg(), nie ponoszą skutków dużego obciążenia, ponieważ nie wymagają informacji, który proces roboczy będzie obsługiwał każde połączenie przychodzące. Gdy serwer wywoła funkcję sendmsg(), deskryptor połączenia przychodzącego i inne dane sterujące zostają umieszczone w kolejce wewnętrznej dla gniazda AF_UNIX. Gdy zadanie procesu roboczego stanie się dostępne, wywoła funkcję recvmsg(), po czym otrzyma pierwszy deskryptor i dane sterujące z kolejki.

Nieaktywne zadanie procesu roboczego

Funkcja givedescriptor() wymaga, aby zadanie procesu roboczego było aktywne, podczas gdy funkcja sendmsg() tego nie wymaga. Do realizacji zadania wywołującego funkcję sendmsg() nie są konieczne żadne informacje 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, za pomocą funkcji sendmsg() wysłać deskryptor przez jedno z gniazd AF_UNIX utworzonych przez funkcję socketpair(), a następnie wywołać funkcję spawn() w celu utworzenia zadania potomnego, które odziedziczy drugie zakończenie pary gniazd. Zadanie potomne wywołuje funkcję recvmsg() w celu otrzymania deskryptora przekazanego przez serwer. Gdy serwer wywoływał funkcję sendmsg(), zadanie potomne nie było aktywne.

Jednoczesne przekazanie kilku deskryptorów

Funkcje givedescriptor() i takedescriptor() umożliwiają przekazywanie deskryptorów pojedynczo. Z kolei funkcje sendmsg() i recvmsg() pozwalają na przekazanie tablicy deskryptorów.

Odsyłacze pokrewne

“Przykład: przekazywanie deskryptorów między procesami” na stronie 97
W tych przykładach przedstawiono sposób projektowania programu serwerowego do obsługi połączeń przychodzących za pomocą funkcji API sendmsg() i recvmsg().

Informacje pokrewne

Funkcja API socketpair() - tworzenie pary gniazd

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

W tym przykładzie przedstawiono typową sytuację, w której można wykorzystać rodzinę adresów AF_INET6.

Sytuacja

Użytkownik jest programistą zatrudnionym w firmie tworzącej aplikacje z użyciem gniazd dla systemu operacyjnego i5/OS. Aby utrzymać przewagę nad konkurencją, firma zamierza opracować pakiet aplikacji używających rodziny adresów AF_INET6, które będą przyjmować połączenia z systemów IPv4 i IPv6. Zadaniem programisty jest napisanie aplikacji przetwarzającej żądania zarówno z węzłów IPv4, jak i IPv6. Programista wie, że system operacyjny i5/OS obsługuje gniazda rodziny adresów AF_INET6, co zapewnia współdziałanie z gniazdami rodziny adresów AF_INET. Wie także, że może się posłużyć formatem adresu IPv6 odwzorowanym z adresu IPv4.

Cele scenariusza

Scenariusz 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.

Wymagania wstępne

Aby opracować aplikację realizującą wymienione cele, należy wykonać następujące czynności:

1. Zainstaluj bibliotekę QSYSINC. Biblioteka ta zawiera pliki nagłówkowe, niezbędne przy kompilowaniu aplikacji używających gniazd.
2. Zainstaluj program licencjonowany ILE C (5761-WDS, opcja 51).
3. Zainstaluj i skonfiguruj kartę Ethernet. Informacje dotyczące opcji sieci Ethernet znajdują się w sekcji Ethernet w Centrum informacyjnym.
4. Skonfiguruj sieć TCP/IP i IPv6. Skorzystaj z informacji o konfigurowaniu TCP/IP i IPv6.

Szczegóły scenariusza

Na poniższym rysunku przedstawiono sieć IPv6, dla której użytkownik tworzy aplikacje obsługujące żądania od klientów IPv6 i IPv4. System operacyjny i5/OS zawiera program, który nasłuchuje żądań od takich klientów i przetwarza je. Sieć tworzą dwie odrębne domeny. Jedna z nich składa się wyłącznie z klientów IPv4, a druga, zdalna - wyłącznie z klientów IPv6. Nazwa domeny to myserver.myco.com. Aplikacja serwerowa przetwarza przychodzące żądania z użyciem rodziny adresów AF_INET6 poprzez wywołanie funkcji API bind() z parametrem in6addr_any.

Odsyłacze pokrewne

“Korzystanie z rodziny adresów AF_INET6” na stronie 27
Gniazda AF_INET6 zapewniają obsługę 128-bitowych (16-bajtowych) struktur adresów protokołu IP w wersji 6 (IPv6). Programiści mogą tworzyć aplikacje korzystające z rodziny adresów AF_INET6, które będą akceptowały połączenia z węzłów klienckich obsługujących zarówno protokół IPv4, jak i IPv6 lub tylko protokół IPv6.

Informacje pokrewne

Ethernet

Konfigurowanie protokołu TCP/IP po raz pierwszy

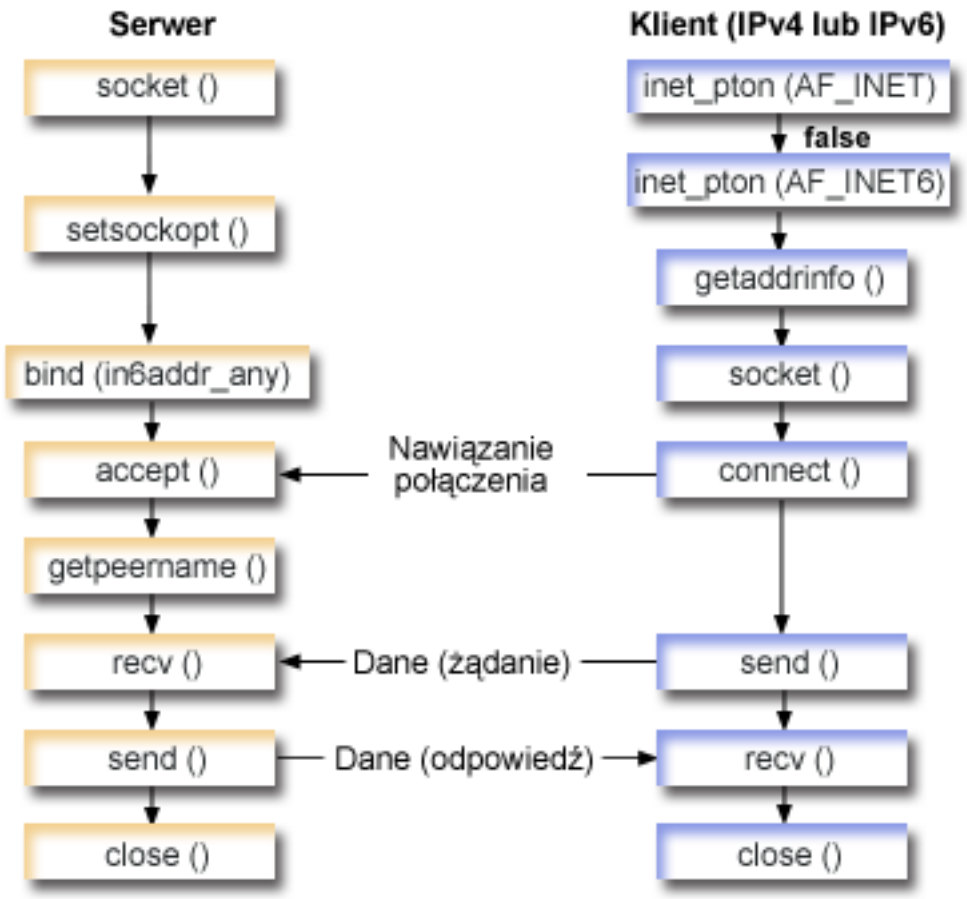
Konfigurowanie protokołu IPv6

Przykład: akceptowanie połączeń od klientów IPv6 i IPv4

W tym programie przykładowym przedstawiono sposób tworzenia modelu serwer/klient, w którym przyjmowane są żądania od aplikacji zarówno w standardzie IPv4 (aplikacje używające gniazd z rodziną adresów AF_INET), jak i IPv6 (aplikacje używające gniazd z rodziną adresów AF_INET6).

Obecnie aplikacje używające gniazd korzystają tylko z 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.

Na poniższym rysunku przedstawiono sposób działania programu:



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

Poniżej opisano poszczególne wywołania funkcji API i ich rolę w aplikacji używającej 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 jest używana rodzina adresów AF_INET6, która obsługuje IPv6, i protokół transportowy TCP (SOCK_STREAM).
2. Funkcja API setsockopt() umożliwia aplikacji ponowne użycie adresu lokalnego w razie zrestartowania serwera przed upływem wymaganego czasu oczekiwania.
3. Funkcja API bind() podaje unikalną nazwę 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 powiązanie to dotyczy zarówno portu IPv4, jak i IPv6).

   Uwaga: Jeśli serwer ma obsługiwać wyłącznie klientów IPv6, można użyć opcji gniazda IPV6_ONLY.
4. Funkcja API listen() pozwala serwerowi na przyjmowanie połączeń przychodzących od klientów. W tym przykładzie programista ustawił wartość kolejki (backlog) na 10. Oznacza to, że system umieści w kolejce pierwsze 10 przychodzących żądań połączenia, kolejne zaś odrzuci.
5. Serwer akceptuje żądanie połączenia przychodzącego za pomocą funkcji API accept(). Wywołanie funkcji API accept() zostanie zablokowane na nieokreślony czas oczekiwania na połączenie przychodzące od klienta IPv4 lub IPv6.

6. Funkcja API getpeername() zwraca do aplikacji adres klienta. Jeśli jest to klient IPv4, zostanie wyświetlony adres IPv6 odwzorowany na adres IPv4.
7. Funkcja API recv() odbiera 250 bajtów danych od klienta. W tym przykładzie klient wysyła 250 bajtów danych. W związku z tym programista może użyć opcji gniazda SO_RCVLOWAT i określić, że funkcja API recv() ma pozostać nieaktywna aż do nadejścia wszystkich 250 bajtów danych.
8. Funkcja API send() odsyła dane do klienta.
9. Funkcja API close() zamyka wszelkie otwarte deskryptory gniazd.

Przebieg zdarzeń w gnieździe: żądania od klientów IPv4 lub IPv6

Uwaga: Tego przykładu klienta można użyć z innymi projektami aplikacji serwera, które akceptują żądania z węzłów IPv4 i IPv6. Z tym przykładem klienta mogą być używane inne projekty serwerów.

1. Wywołanie funkcji API inet_pton() przekształca tekstową postać adresu w binarną. W tym przykładzie funkcja jest wywoływana 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, to funkcja getaddrinfo() nie powinna dokonywać tłumaczenia nazwy. W przeciwnym razie jest to nazwa hosta, którą należy przetłumaczyć podczas wywołania funkcji getaddrinfo().
2. Funkcja API getaddrinfo() pobiera informacje o adresie potrzebne w późniejszych wywołaniach funkcji API socket() i connect().
3. Funkcja API socket() zwraca deskryptor gniazda reprezentujący punkt końcowy. Ponadto instrukcja ta na podstawie informacji zwróconych przez funkcję API getaddrinfo() identyfikuje rodzinę adresów, typ gniazda oraz protokół.
4. Funkcja API connect() nawiązuje połączenie z serwerem niezależnie od tego, czy jest to serwer IPv4, czy IPv6.
5. Funkcja API send() wysyła do serwera żądanie danych.
6. Funkcja API recv() odbiera dane z aplikacji serwera.
7. Funkcja API close() zamyka wszelkie otwarte deskryptory gniazd.

Poniższy kod przykładowy przedstawia aplikację serwera w tym scenariuszu.

Uwaga: Korzystając z przykładów, użytkownik wyraża zgodę na warunki podane w sekcji “Licencja na kod oraz Informacje dotyczące kodu” na stronie 199.

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

/*********************************************************************/
/* Stałe używane przez 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 reprezentujący punkt */
   /* końcowy. Aby przygotować aplikację na odbieranie połączeń       */
   /* przychodzących, należy pobrać gniazdo dla AF_INET6.             */
   /*******************************************************************/
   if ((sd = socket(AF_INET6, SOCK_STREAM, 0)) < 0)
   {
      perror("Niepowodzenie funkcji socket()");
      break;
   }

   /*******************************************************************/
   /* Funkcja setsockopt() umożliwia ponowne użycie adresu lokalnego  */
   /* w razie zrestartowania serwera przed upływem wymaganego czasu   */
   /* oczekiwania.                                                    */
   /*******************************************************************/
   if (setsockopt(sd, SOL_SOCKET, SO_REUSEADDR,
                  (char *)&on,sizeof(on)) < 0)
   {
      perror("Niepowodzenie funkcji setsockopt(SO_REUSEADDR)");
      break;
   }

   /*******************************************************************/
   /* Po utworzeniu deskryptora gniazda funkcja bind() pobiera        */
   /* unikalną nazwę gniazda. W tym przykładzie użytkownik ustawia    */
   /* adres na in6addr_any, co (domyślnie) umożliwia nawiązywanie     */
   /* połączeń z dowolnego klienta IPv4 lub IPv6, który określi port  */
   /* 3005. Oznacza to, że powiązanie to dotyczy zarówno portu IPv4,  */
   /* jak 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 IPv4. Stała symboliczna IN6ADDR_ANY_INIT */
   /* również istnieje, ale może zostać użyta tylko do zainicjowania  */
   /* struktury in6_addr podczas deklarowania (nie 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 funkcji bind()");
      break;
   }

   /*******************************************************************/
   /* Funkcja listen() pozwala serwerowi na przyjmowanie połączeń     */
```

```
/* przychodzących od klienta. W tym przykładzie kolejka (backlog)   */
/* ma wartość 10. Oznacza to, że system umieści w kolejce pierwsze  */
/* 10 przychodzących żądań połączenia,                              */
/* kolejne zaś odrzuci.                                             */
/********************************************************************/
if (listen(sd, 10) < 0)
{
   perror("Niepowodzenie funkcji listen()");
   break;
}

printf("Gotowy do nawiązania połączenia z klientem.\n");

/********************************************************************/
/* Serwer zaakceptuje żądanie połączenia przychodzącego za pomocą   */
/* funkcji accept(). Wywołanie accept() zostanie zablokowane        */
/* na nieokreślony czas oczekiwania na połączenie przychodzące od   */
/* klienta IPv4 lub IPv6.                                           */
/********************************************************************/
if ((sdconn = accept(sd, NULL, NULL)) < 0)
{
   perror("Niepowodzenie funkcji accept()");
   break;
}
else
{
   /*****************************************************************/
   /* Wyświetla adres klienta. Jeśli jest to klient IPv6, zostanie  */
   /* wyświetlony adres IPv6 odwzorowany na adres                   */
   /* IPv4.                                                         */
   /*****************************************************************/
   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.    */
/* W związku z tym można użyć opcji gniazda SO_RCVLOWAT i określić, */
/* że funkcja recv() ma pozostać nieaktywna aż do nadejścia         */
/* wszystkich 250 bajtów danych.                                    */
/********************************************************************/
if (setsockopt(sdconn, SOL_SOCKET, SO_RCVLOWAT,
               (char *)&rcdsize,sizeof(rcdsize)) < 0)
{
   perror("Niepowodzenie funkcji setsockopt(SO_RCVLOWAT)");
   break;
}

/********************************************************************/
/* Odbierz 250 bajtów od klienta                                    */
/********************************************************************/
rc = recv(sdconn, buffer, sizeof(buffer), 0);
if (rc < 0)
{
   perror("Niepowodzenie funkcji 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;
      }

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

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

   } while (FALSE);

   /*******************************************************************/
   /* Zamknij wszystkie otwarte deskryptory gniazd                    */
   /*******************************************************************/
   if (sd != -1)
      close(sd);
   if (sdconn != -1)
      close(sdconn);
}
```

Odsyłacze pokrewne

“Przykłady: projekty aplikacji zorientowanych na połączenie” na stronie 87
System udostępnia kilka metod projektowania serwera używającego gniazd zorientowanego na połączenia. Do tworzenia własnych programów zorientowanych na połączenie można użyć poniższych programów przykładowych.

“Przykład: klient IPv4 lub IPv6”
Tego programu przykładowego można użyć razem z aplikacją serwera, która akceptuje żądania od klientów IPv4 i IPv6.

“Przykład: ogólny program klienta” na stronie 109
W przykładzie użyto kodu typowych zadań klienta. Zadanie klienta używa funkcji socket(), connect(), send(), recv() i close().

Informacje pokrewne

Funkcja API socket() - tworzenie gniazd

Funkcja API setsockopt_ioctl() - ustawianie opcji gniazd

Funkcja API bind() - ustawianie lokalnego adresu gniazda

Funkcja API listen() - nasłuchiwanie przychodzących żądań połączenia

Funkcja API accept() - oczekiwanie na żądanie i nawiązywanie połączenia

Funkcja API getpeername() - wczytywanie docelowego adresu gniazda

Funkcja API recv() - odbieranie danych

Funkcja API send() - wysyłanie danych

Funkcja API close() - zamykanie deskryptora gniazda lub pliku

inet_pton()

Funkcja API getaddrinfo() - pobieranie informacji o adresie

Funkcja API connect() - nawiązywanie połączenia lub ustanawianie adresu docelowego

Przykład: klient IPv4 lub IPv6

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

Uwaga: Korzystając z przykładów, użytkownik wyraża zgodę na warunki podane w sekcji “Licencja na kod oraz Informacje dotyczące kodu” na stronie 199.

```
/*********************************************************************/
/* Klient IPv4 lub IPv6.                                             */
/*********************************************************************/

/*********************************************************************/
/* Pliki nagłówkowe wymagane przez program przykładowy               */
/*********************************************************************/
#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 program                                       */
/*********************************************************************/
#define BUFFER_LENGTH    250
#define FALSE              0
#define SERVER_NAME     "ServerHostName"

/* Przekaż 1 parametr, będący albo         */
/* adresem, albo nazwą hosta serwera lub   */
/* ustaw nazwę serwera SERVER_NAME         */
/* w makrze #define                        */
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 */
      /* API inet_pton(), która przekształca tekstową postać adresu   */
      /* w binarną. Jeśli adres jest liczbowy, funkcja API getaddrinfo()  */
      /* nie powinna wykonywać tłumaczenia 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 funkcji getaddrinfo()");
     break;
  }

  /*********************************************************************/
  /* Funkcja socket() zwraca deskryptor gniazda reprezentujący punkt  */
  /* końcowy. Instrukcja ta informuje również, że dla tego gniazda    */
  /* zostały określone 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 funkcji socket()");
     break;
  }
  /*********************************************************************/
  /* Aby nawiązać połączenie z serwerem, użyj funkcji 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 adresie, mogą zostać wypróbowane kolejne            */
     /* adresy na liście.                                             */
     /*******************************************************************/
     perror("Niepowodzenie funkcji connect()");
     break;
  }

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

```
      }

      /*********************************************************************/
      /* W tym przykładzie wiadomo, że serwer odpowie wysłaniem tych      */
      /* samych 250 bajtów, które wysłano. 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 użyto innej metody. Ponieważ te 250       */
      /* bajtów danych może być przysłane w oddzielnych pakietach,        */
      /* funkcja recv() będzie uruchamiana wielokrotnie aż do             */
      /* nadejścia wszystkich 250 bajtów.                                 */
      /*********************************************************************/
      while (bytesReceived < BUFFER_LENGTH)
      {
         rc = recv(sd, & buffer[bytesReceived],
                   BUFFER_LENGTH - bytesReceived, 0);
         if (rc < 0)
         {
            perror("Niepowodzenie funkcji recv()");
            break;
         }
         else if (rc == 0)
         {
            printf("Serwer zamknął połączenie\n");
            break;
         }

         /*********************************************************************/
         /* Zwiększ liczbę bajtów dotychczas otrzymanych                      */
         /*********************************************************************/
         bytesReceived += rc;
      }

   } while (FALSE);

   /*********************************************************************/
   /* Zamknij wszystkie otwarte deskryptory gniazd                      */
   /*********************************************************************/
   if (sd != -1)
      close(sd);
   /*********************************************************************/
   /* Zwolnienie wszystkich rezultatów działania funkcji getaddrinfo    */
   /*********************************************************************/
   if (res != NULL)
      freeaddrinfo(res);
}
```

Odsyłacze pokrewne

“Przykład: akceptowanie połączeń od klientów IPv6 i IPv4” na stronie 76
W tym programie przykładowym przedstawiono sposób tworzenia modelu serwer/klient, w którym przyjmowane są żądania od aplikacji zarówno w standardzie IPv4 (aplikacje używające gniazd z rodziną adresów AF_INET), jak i IPv6 (aplikacje używające gniazd z rodziną adresów AF_INET6).

“Przykład: ogólny program klienta” na stronie 109
W przykładzie użyto kodu typowych zadań klienta. Zadanie klienta używa funkcji socket(), connect(), send(), recv() i close().

Zalecenia dotyczące projektowania aplikacji używających gniazd

Przed przystąpieniem do pracy z aplikacją używającą gniazd należy ocenić jej wymagania funkcjonalne, cele i potrzeby. Należy również rozważyć wymagania aplikacji dotyczące wydajności oraz jej wpływ na zasoby systemu.

Poniższa lista zaleceń może pomóc rozwiązać wymienione kwestie i wskazać bardziej efektywne metody wykorzystania gniazd i projektowania aplikacji używających gniazd:

Tabela 17. Projektowanie aplikacji używających gniazd

Zalecenie	Przyczyna	Najlepsze zastosowanie
Użycie asynchronicznych operacji we/wy	Zamiast konwencjonalnego modelu opartego na funkcji select() preferowane jest użycie asynchronicznych operacji we/wy w modelu serwera z obsługą wątków.	Aplikacje serwera używające gniazd, przeznaczone do współbieżnej obsługi wielu klientów.
Jeśli używasz asynchronicznych operacji we/wy, dostosuj liczbę wątków w procesie do optymalnej liczby klientów wymagających przetwarzania.	Zbyt mała liczba wątków sprawia, że w przypadku niektórych klientów czas oczekiwania na obsługę może zostać przekroczony. Zbyt duża liczba wątków sprawia, że niektóre zasoby systemowe nie są wykorzystywane w sposób efektywny. **Uwaga:** Lepiej jest, 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 znaczników końcowych we wszystkich operacjach rozpoczynających asynchroniczne operacje we/wy.	Jeśli operacja została wykonana synchronicznie, pozwala to uniknąć dodatkowego obciążenia związanego z przejściem do portu zakończenia.	Aplikacje używające gniazd, korzystające z asynchronicznych operacji we/wy.
Korzystaj z funkcji send() i recv() zamiast read() i write().	Funkcje send() i recv() mają nieco poprawioną wydajność i możliwość obsługi serwisowej w stosunku do funkcji read() i write().	Dowolny program używający gniazd, który rozróżnia używanie deskryptorów gniazd od deskryptorów plików.
Używaj opcji odbierania SO_RCVLOWAT, aby uniknąć zapętlenia podczas operacji odbierania, dopóki nie zostaną przesłane wszystkie dane.	Dzięki temu aplikacja może oczekiwać na minimalną ilość danych do odebrania przez gniazdo przed spełnieniem warunku zablokowanej operacji odbioru.	Dowolna aplikacja używająca gniazd, która odbiera dane.
Używaj opcji MSG_WAITALL, aby uniknąć zapętlenia podczas operacji odbierania, dopóki nie zostaną przesłane wszystkie dane.	Dzięki temu aplikacja może oczekiwać na całą zawartość buforu operacji odbierania przed spełnieniem warunku zablokowanej operacji odbioru.	Dowolna aplikacja używająca gniazd, która odbiera dane i wie z góry, jakiej ilości danych ma oczekiwać.
Używaj funkcji API sendmsg() i recvmsg() zamiast funkcji givedescriptor() i takedescriptor().	Zalety tego rozwiązania zostały szczegółowo opisane w sekcji “Przekazywanie deskryptorów między procesami - funkcje sendmsg() i recvmsg()” na stronie 73.	Dowolna aplikacja używająca gniazd, która przekazuje deskryptory gniazd lub plików między procesami.
Podczas używania funkcji select() staraj się unikać dużej liczby deskryptorów w zestawie operacji odczytu, zapisu lub w zestawie wyjątków. **Uwaga:** W przypadku dużej liczby deskryptorów używanych w celu przetwarzania funkcji select() należy kierować się zaleceniem dotyczącym używania asynchronicznych operacji we/wy przedstawionym powyżej.	W przypadku dużej liczby deskryptorów w zestawie operacji odczytu, zapisu lub w zestawie wyjątków przy każdym wywołaniu funkcji select() jest wykonywanych wiele powtarzających się czynności. Po wykonaniu funkcji select() należy wykonać bieżącą 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 deskryptorów aktywnych dla funkcji select().

Tabela 17. Projektowanie aplikacji używających gniazd (kontynuacja)

Zalecenie	Przyczyna	Najlepsze zastosowanie
Przed wywołaniem funkcji select() zapisz zestawy operacji odczytu, zapisu lub zestawy wyjątków, aby uniknąć odbudowywania zestawów podczas każdego wywołania funkcji select().	Pozwala to uniknąć dodatkowych czynności związanych z przebudowywaniem zestawów operacji odczytu, zapisu i zestawów wyjątków za każdym razem, gdy jest planowane wywołanie funkcji 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, konieczne może być użycie jako zegara funkcji select(). W takim przypadku ustaw maksymalną liczbę deskryptorów na 0, a zestawy operacji odczytu, zapisu i zestawy wyjątków na NULL.	Pozwoli to na uzyskiwanie lepszych odpowiedzi zegara i zmniejszy obciążenie systemu.	Dowolna aplikacja używająca gniazd, w której funkcja select() jest stosowana wyłącznie jako zegar.
Jeśli aplikacja używająca gniazd zwiększyła maksymalną liczbę deskryptorów gniazd i plików dozwolonych dla pojedynczego procesu za pomocą funkcji DosSetRelMaxFH() oraz jeśli w tej samej aplikacji jest używana funkcja select(), to należy zwrócić 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, w których 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.	Pozwala to zminimalizować liczbę wywołań funkcji select(), kiedy dane są wciąż dostępne do przetwarzania lub odczytu dla danego deskryptora.	Dowolna aplikacja używająca gniazd, w której jest stosowana funkcja select().
Podczas wykorzystywania przetwarzania funkcji select() określ tylko niezbędne zestawy.	Większość aplikacji nie potrzebuje określania zestawu wyjątków ani zestawu operacji zapisu.	Dowolna aplikacja używająca gniazd, w której jest stosowana funkcja select().
Używaj funkcji API GSKit zamiast funkcji API SSL_.	Zarówno funkcje API GSKit, jak i funkcje SSL_ systemu i5/OS umożliwiają tworzenie aplikacji używających gniazd chronionych typu SOCK_STREAM z rodziny adresów AF_INET lub AF_INET6. Jako że funkcje API GSKit są obsługiwane we wszystkich systemach IBM, są preferowanymi funkcjami API do ochrony aplikacji. Funkcje API SSL_ występują tylko w systemie operacyjnym i5/OS.	Każda aplikacja używająca gniazd, która musi obsługiwać przetwarzanie SSL lub TLS.
Nie używaj sygnałów.	Z sygnałami jest związane duże obciążenie (na wszystkich platformach, nie tylko na platformie System i). Lepiej zaprojektować aplikację, która będzie używać asynchronicznych operacji we/wy lub funkcji select().	Dowolna aplikacja używająca gniazd, która korzysta z sygnałów.

Tabela 17. Projektowanie aplikacji używających gniazd (kontynuacja)

Zalecenie	Przyczyna	Najlepsze zastosowanie
W miarę możliwości używaj procedur niezależnych od protokołu, takich jak inet_ntop(), inet_pton(), getaddrinfo() i getnameinfo().	Nawet jeśli aplikacja nie ma obsługiwać adresu IPv6, warto używać tych funkcji API (zamiast inet_ntoa(), inet_addr(), gethostbyname() i gethostbyaddr()), aby ułatwić późniejszą migrację.	Dowolna aplikacja z rodziny adresów AF_INET lub AF_INET6 korzystająca z procedur sieciowych.
Użyj sockaddr_storage do zadeklarowania obszaru pamięci masowej dla dowolnego adresu z tej rodziny adresów.	Upraszcza pisanie kodu przenośnego między wieloma rodzinami adresów i platformami. Deklaruje wystarczającą ilość pamięci masowej do pomieszczenia jak największej rodziny adresów i zapewnia poprawne wyrównanie adresu.	Dowolna aplikacja używająca gniazd, która przechowuje adresy.

Pojęcia pokrewne

"Asynchroniczne operacje we/wy" na stronie 42
Funkcje API asynchronicznych operacji we/wy udostępniają metodę realizacji wątkowych modeli klient/serwer w celu zapewnienia wysoko współbieżnych operacji we/wy z efektywnym wykorzystaniem pamięci.

Odsyłacze pokrewne

"Przykład: korzystanie z asynchronicznych operacji we/wy" na stronie 112
Aplikacja tworzy port zakończenia operacji we/wy za pomocą funkcji API QsoCreateIOCompletionPort(). Funkcja ta zwraca uchwyt, za pomocą którego można zaplanować asynchroniczne żądania we/wy i oczekiwać na ich zakończenie.

"Przykład: nieblokujące operacje we/wy i funkcja select()" na stronie 155
W tym przykładzie przedstawiono aplikację serwera, która wykorzystuje nieblokujące operacje we/wy i funkcję API select().

"Przykład: przekazywanie deskryptorów między procesami" na stronie 97
W tych przykładach przedstawiono sposób projektowania programu serwerowego do obsługi połączeń przychodzących za pomocą funkcji API sendmsg() i recvmsg().

Informacje pokrewne

DosSetRelMaxFH()

Przykłady: projekty aplikacji używających gniazd

Programy przedstawione w tych przykładach stanowią ilustrację zaawansowanych koncepcji korzystania z gniazd. Można ich użyć do tworzenia własnych aplikacji, które będą realizowały podobne zadania.

Przykładom towarzyszą rysunki i lista wywołań, które ilustrują przebieg zdarzeń w poszczególnych aplikacjach. Można skorzystać z interaktywnych możliwości narzędzia Xsockets, wypróbować niektóre z funkcji API użytych w programach lub wprowadzić zmiany specyficzne dla własnego środowiska.

Przykłady: projekty aplikacji zorientowanych na połączenie

System udostępnia kilka metod projektowania serwera używającego gniazd zorientowanego na połączenia. Do tworzenia własnych programów zorientowanych na połączenie można użyć poniższych programów przykładowych.

Mimo że możliwe są inne projekty serwerów używających gniazd, poniższe przykłady są najpowszechniejsze.

Serwer iteracyjny

W przykładzie serwera iteracyjnego jedno zadanie serwera obsługuje wszystkie połączenia przychodzące, a przepływ wszystkich danych następuje w zadaniach klienta. Po zakończeniu działania funkcji API accept() serwer obsługuje całą transakcję. Ten serwer jest najprostszy do zaprogramowania, ale wiążą się z nim pewne problemy. Podczas gdy serwer obsługuje żądanie od określonego klienta, inne klienty mogą próbować się z nim połączyć. Żądania te zapełniają

kolejkę (backlog) funkcji listen() i ostatecznie niektóre z nich zostają odrzucone.

Serwer współbieżny

W projektach serwerów współbieżnych system używa wielu zadań i wątków do obsługi przychodzących żądań połączenia. Serwery współbieżne są używane wtedy, gdy może się z nimi łączyć jednocześnie wiele klientów.

Jeśli w sieci jest wiele klientów współbieżnych, to zaleca się użycie funkcji API gniazd obsługujących asynchroniczne operacje we/wy. Funkcje te zapewniają największą wydajność w sieciach z wieloma klientami współbieżnymi.

- Serwer spawn() i proces roboczy spawn()

  Za pomocą funkcji API spawn() tworzone jest nowe zadanie do obsługi każdego żądania przychodzącego. Po zakończeniu działania funkcji spawn() serwer może oczekiwać przy funkcji API accept() na odebranie następnego połączenia przychodzącego.

  Jedynym problemem związanym z tym projektem serwera jest narzut związany z tworzeniem nowego zadania przy każdym połączeniu. Można tego uniknąć dzięki użyciu zadań prestartu. Zamiast tworzenia nowego zadania dla każdego odebranego połączenia następuje przekazanie połączenia przychodzącego do zadania, które jest już aktywne. We wszystkich pozostałych przykładach w tej sekcji są używane zadania prestartu.
- Serwer spawn() i proces roboczy spawn()

  Funkcje sendmsg() i recvmsg() są używane do obsługi połączeń przychodzących. Przy pierwszym uruchomieniu zadania serwera dokonuje on prestartu wszystkich zadań procesu roboczego.
- Wiele serwerów accept() i wiele procesów roboczych accept()

  W poprzednich funkcjach API zadanie procesu roboczego nie wykonywało żadnych czynności, dopóki serwer nie odebrał żądania połączenia przychodzącego. Dzięki użyciu wielu funkcji API accept() każde zadanie procesu roboczego można zamienić w serwer iteracyjny. Zadanie serwera nadal wywołuje funkcje API socket(), bind() i listen(). Po zakończeniu działania funkcji listen() serwer tworzy poszczególne zadania procesów roboczych i przydziela każdemu z nich gniazdo do nasłuchu. Następnie wszystkie zadania procesów roboczych wywołują funkcję API accept(). Kiedy klient próbuje ustanowić połączenie z serwerem, następuje zakończenie tylko jednego wywołania funkcji accept() i ten proces roboczy obsługuje połączenie.

  Pojęcia pokrewne

  “Asynchroniczne operacje we/wy” na stronie 42
  Funkcje API asynchronicznych operacji we/wy udostępniają metodę realizacji wątkowych modeli klient/serwer w celu zapewnienia wysoko współbieżnych operacji we/wy z efektywnym wykorzystaniem pamięci.

  Odsyłacze pokrewne

  “Przykład: akceptowanie połączeń od klientów IPv6 i IPv4” na stronie 76
  W tym programie przykładowym przedstawiono sposób tworzenia modelu serwer/klient, w którym przyjmowane są żądania od aplikacji zarówno w standardzie IPv4 (aplikacje używające gniazd z rodziną adresów AF_INET), jak i IPv6 (aplikacje używające gniazd z rodziną adresów AF_INET6).

  “Przykład: korzystanie z asynchronicznych operacji we/wy” na stronie 112
  Aplikacja tworzy port zakończenia operacji we/wy za pomocą funkcji API QsoCreateIOCompletionPort(). Funkcja ta zwraca uchwyt, za pomocą którego można zaplanować asynchroniczne żądania we/wy i oczekiwać na ich zakończenie.

  “Przykład: ogólny program klienta” na stronie 109
  W przykładzie użyto kodu typowych zadań klienta. Zadanie klienta używa funkcji socket(), connect(), send(), recv() i close().

  Informacje pokrewne

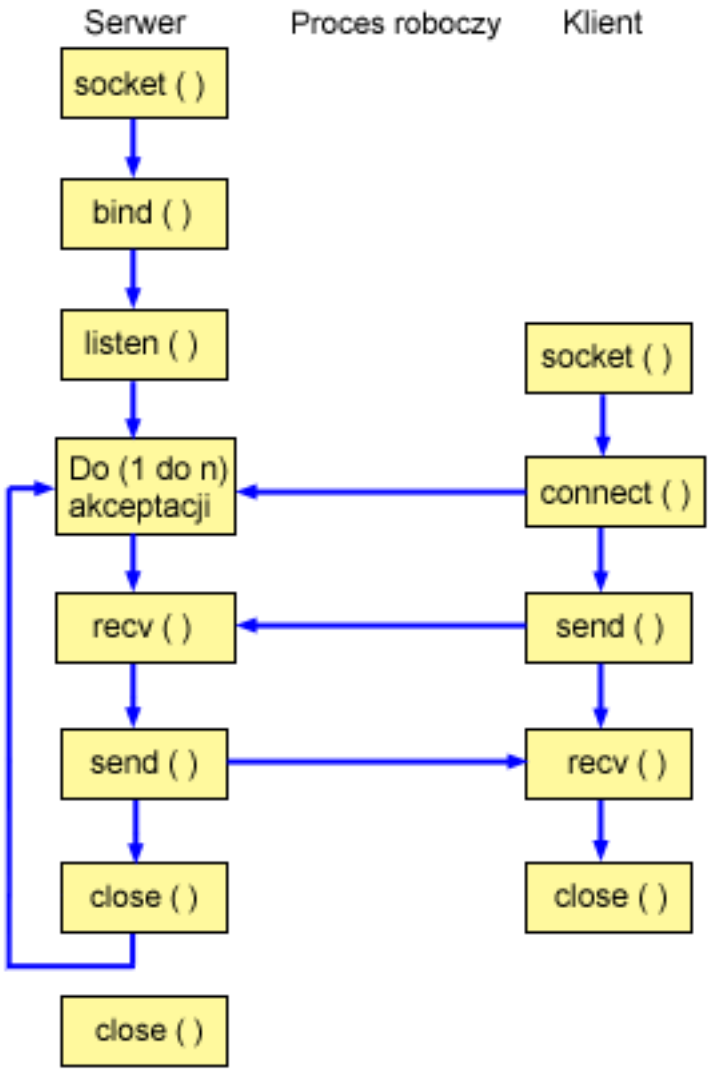
  Funkcja API accept() - oczekiwanie na żądanie i nawiązywanie połączenia

  spawn()

Przykład: pisanie programu serwera iteracyjnego

W tym przykładzie przedstawiono sposób tworzenia pojedynczego zadania serwera, które obsługuje wszystkie połączenia przychodzące. Po zakończeniu działania funkcji API accept() serwer obsługuje całą transakcję.

Na poniższym rysunku przedstawiono współdziałanie zadań serwera i klienta współdziałają ze sobą, gdy w systemie jest używany program serwera iteracyjnego.



Przebieg zdarzeń w gnieździe: serwer iteracyjny

Poniżej wyjaśniono sekwencję wywołań funkcji API gniazd, przedstawionych na rysunku. Opisano także relacje między aplikacją serwera a aplikacją procesu roboczego. Każdy zbiór przepływów zawiera odsyłacze do uwag dotyczących użycia poszczególnych funkcji API. Więcej informacji na temat użycia tych funkcji API można uzyskać za pomocą wymienionych odsyłaczy. Poniżej opisano kolejność wywołań funkcji API w aplikacji serwera iteracyjnego:

1. Funkcja API socket() zwraca deskryptor gniazda reprezentujący punkt końcowy. Instrukcja ta informuje również, że dla tego gniazda zostanie użyta rodzina adresów INET (Internet Protocol) i protokół transportowy TCP (SOCK_STREAM).
2. Po utworzeniu deskryptora gniazda funkcja API bind() pobiera unikalną nazwę gniazda.
3. Funkcja API listen() pozwala serwerowi na przyjmowanie połączeń przychodzących od klientów.
4. Serwer akceptuje żądanie połączenia przychodzącego za pomocą funkcji API accept(). Wywołanie funkcji API accept() zostanie zablokowane na nieokreślony czas oczekiwania na połączenie przychodzące.
5. Funkcja API recv() odbiera dane z aplikacji klienta.
6. Funkcja API send() odsyła dane do klienta.

7. Funkcja API close() zamyka wszelkie otwarte deskryptory gniazd.

Uwaga: Korzystając z przykładów, użytkownik wyraża zgodę na warunki podane w sekcji “Licencja na kod oraz Informacje dotyczące kodu” na stronie 199.

```
/*****************************************************************************/
/* Aplikacja tworzy program 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 strumieniowe AF_INET do        */
   /* odbierania połączeń przychodzących            */
   /*************************************************/
   listen_sd = socket(AF_INET, SOCK_STREAM, 0);
   if (listen_sd < 0)
   {
      perror("Niepowodzenie funkcji socket()");
      exit(-1);
   }

   /*************************************************/
   /* Pozwól na ponowne użycie deskryptora gniazda  */
   /*************************************************/
   rc = setsockopt(listen_sd,
                   SOL_SOCKET,  SO_REUSEADDR,
                   (char *)&on, sizeof(on));
   if (rc < 0)
   {
      perror("Niepowodzenie funkcji 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 funkcji bind()");
```

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

/*********************************************/
/* Ustaw parametr backlog funkcji listen.    */
/*********************************************/
rc = listen(listen_sd, 5);
if (rc < 0)
{
   perror("Niepowodzenie funkcji 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++)
{
   /*********************************************/
   /* Czekaj na połączenie przychodzące         */
   /*********************************************/
   printf("Iteracja: %d\n", i+1);
   printf("  Oczekiwanie na wywołanie funkcji accept()\n");
   accept_sd = accept(listen_sd, NULL, NULL);
   if (accept_sd < 0)
   {
      perror("Niepowodzenie funkcji accept()");
      close(listen_sd);
      exit(-1);
   }
   printf(" Wywołanie funkcji accept zakończone powodzeniem\n");

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

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

```
      /*********************************************/
      /* Zamknij połączenie przychodzące           */
      /*********************************************/
      close(accept_sd);
   }

   /*********************************************/
   /* Zamknij gniazdo nasłuchujące              */
   /*********************************************/
   close(listen_sd);
}
```

Odsyłacze pokrewne

“Przykład: ogólny program klienta” na stronie 109
W przykładzie użyto kodu typowych zadań klienta. Zadanie klienta używa funkcji socket(), connect(), send(), recv() i close().

Informacje pokrewne

Funkcja API recv() - odbieranie danych

Funkcja API bind() - ustawianie lokalnego adresu gniazda

Funkcja API socket() - tworzenie gniazd

Funkcja API listen() - nasłuchiwanie przychodzących żądań połączenia

Funkcja API accept() - oczekiwanie na żądanie i nawiązywanie połączenia

Funkcja API send() - wysyłanie danych

Funkcja API close() - zamykanie deskryptora gniazda lub pliku

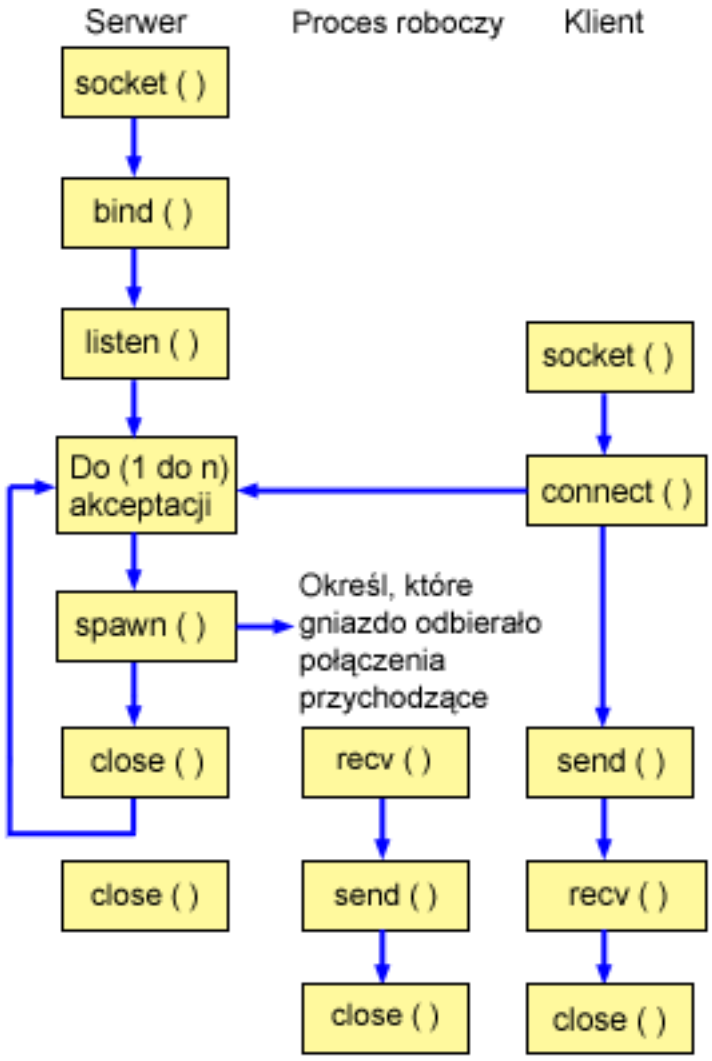
Przykład: używanie funkcji API spawn() do tworzenia procesów potomnych

W tym przykładzie przedstawiono sposób użycia przez program serwera funkcji API spawn() do utworzenia procesu potomnego, który dziedziczy deskryptor gniazda po procesie macierzystym.

Zadanie serwera oczekuje na połączenie przychodzące, a następnie wywołuje funkcję API spawn() w celu utworzenia procesów potomnych do obsługi tego połączenia. Proces potomny utworzony funkcją API spawn() dziedziczy następujące atrybuty:

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

Na poniższym rysunku przedstawiono współdziałanie zadań serwera, procesu roboczego i klienta, gdy w systemie jest wykorzystywany program serwera używającego funkcji spawn().



Przebieg zdarzeń w gnieździe: serwer używający funkcji spawn() do przyjmowania i przetwarzania żądań

Poniżej wyjaśniono sekwencję wywołań funkcji API gniazd, przedstawionych na rysunku. Opisano także relacje między aplikacją serwera a przykładowym procesem roboczym. Każdy zbiór przepływów zawiera odsyłacze do uwag dotyczących użycia poszczególnych funkcji API. Więcej informacji na temat użycia tych funkcji API można uzyskać za pomocą wymienionych odsyłaczy. W pierwszym przykładzie wykorzystano następujące wywołania funkcji gniazd do tworzenia procesu potomnego za pomocą funkcji API spawn():

1. Funkcja API socket() zwraca deskryptor gniazda reprezentujący punkt końcowy. Instrukcja ta informuje również, że dla tego gniazda zostanie użyta rodzina adresów INET (Internet Protocol) i protokół transportowy TCP (SOCK_STREAM).
2. Po utworzeniu deskryptora gniazda funkcja API bind() pobiera unikalną nazwę gniazda.
3. Funkcja API listen() pozwala serwerowi na przyjmowanie połączeń przychodzących od klientów.
4. Serwer akceptuje żądanie połączenia przychodzącego za pomocą funkcji API accept(). Wywołanie funkcji API accept() zostanie zablokowane na nieokreślony czas oczekiwania na połączenie przychodzące.
5. Funkcja API spawn() inicjuje parametry zadania roboczego do obsługi żądań przychodzących. W tym przykładzie deskryptor gniazda nowego połączenia jest w programie potomnym odwzorowywany na deskryptor 0.

6. W tym przykładzie pierwsze wywołanie funkcji API close() zamyka deskryptor gniazda nasłuchującego. Drugie wywołanie funkcji close() zamyka gniazdo akceptujące.

Przebieg zdarzeń w gnieździe: zadanie procesu roboczego utworzone za pomocą funkcji spawn()

W drugim przykładzie jest wykorzystywana następująca sekwencja wywołań funkcji API:

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

Odsyłacze pokrewne

"Przykład: ogólny program klienta" na stronie 109
W przykładzie użyto kodu typowych zadań klienta. Zadanie klienta używa funkcji socket(), connect(), send(), recv() i close().

Informacje pokrewne

spawn()

Funkcja API bind() - ustawianie lokalnego adresu gniazda

Funkcja API socket() - tworzenie gniazd

Funkcja API listen() - nasłuchiwanie przychodzących żądań połączenia

Funkcja API accept() - oczekiwanie na żądanie i nawiązywanie połączenia

Funkcja API close() - zamykanie deskryptora gniazda lub pliku

Funkcja API send() - wysyłanie danych

Funkcja API recv() - odbieranie danych

Przykład: tworzenie serwera używającego funkcji API spawn():

W tym przykładzie przedstawiono sposób użycia funkcji API spawn() do tworzenia procesu potomnego, który dziedziczy deskryptor gniazda od procesu macierzystego.

Uwaga: Korzystając z przykładów, użytkownik wyraża zgodę na warunki podane w sekcji "Licencja na kod oraz Informacje dotyczące kodu" na stronie 199.

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

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

#define SERVER_PORT 12345

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

   /*********************************************/
   /* Jeśli podano argument, użyj go do sterowania  */
   /* liczbą połączeń przychodzących                */
   /*********************************************/
```

```
if (argc >= 2)
   num = atoi(argv[1]);
else
   num = 1;

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

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

/*************************************************/
/* Ustaw parametr backlog funkcji listen.        */
/*************************************************/
rc = listen(listen_sd, 5);
if (rc < 0)
{
   perror("Niepowodzenie funkcji 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++)
{
```

```
      /*********************************************/
      /* Czekaj na połączenie przychodzące         */
      /*********************************************/
      printf("Iteracja: %d\n", i+1);
      printf("  Oczekiwanie na wywołanie funkcji accept()\n");
      accept_sd = accept(listen_sd, NULL, NULL);
      if (accept_sd < 0)
      {
         perror("Niepowodzenie funkcji accept()");
         close(listen_sd);
         exit(-1);
      }
      printf(" Wywołanie funkcji accept zakończone powodzeniem\n");

      /*********************************************/
      /* Zainicjuj parametry funkcji 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 funkcji spawn()");
         close(listen_sd);
         close(accept_sd);
         exit(-1);
      }
      printf(" wywołanie funkcji spawn zakończone powodzeniem\n");

      /*********************************************/
      /* Zamyka połączenie przychodzące, ponieważ  */
      /* zostało ono przekazane do obsługi         */
      /* procesowi roboczemu.                      */
      /*********************************************/
      close(accept_sd);
   }

   /*********************************************/
   /* Zamknij gniazdo nasłuchujące              */
   /*********************************************/
   close(listen_sd);
}
```

Odsyłacze pokrewne

“Przykład: umożliwienie przekazania buforu danych do zadania procesu roboczego”
W przykładzie przedstawiono kod umożliwiający przekazanie buforu danych z zadania klienta do zadania procesu roboczego, a następnie jego odesłanie.

Przykład: umożliwienie przekazania buforu danych do zadania procesu roboczego:

W przykładzie przedstawiono kod umożliwiający przekazanie buforu danych z zadania klienta do zadania procesu roboczego, a następnie jego odesłanie.

Uwaga: Korzystając z przykładów, użytkownik wyraża zgodę na warunki podane w sekcji “Licencja na kod oraz Informacje dotyczące kodu” na stronie 199.

```
/*********************************************************************/
/* 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;

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

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

   /************************************************/
   /* Zamknij połączenie przychodzące               */
   /************************************************/
   close(sockfd);

}
```

Odsyłacze pokrewne

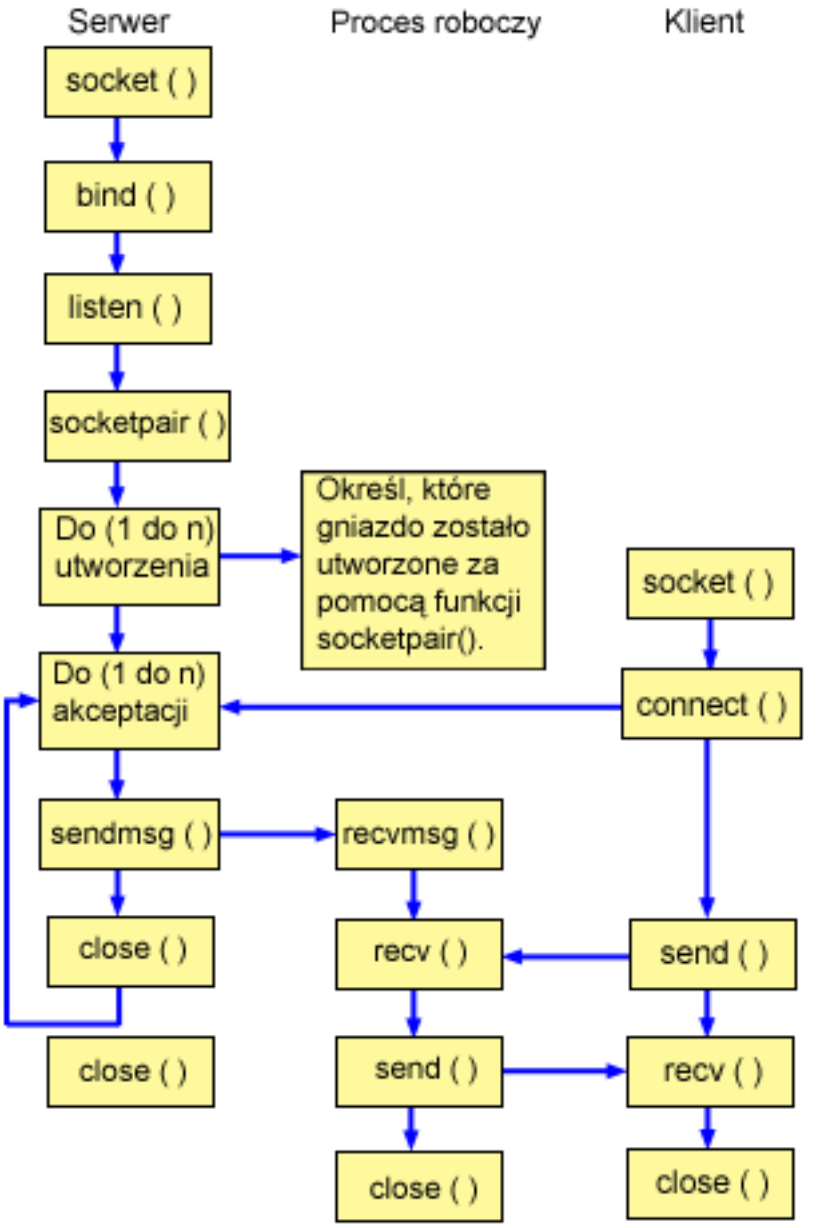
“Przykład: tworzenie serwera używającego funkcji API spawn()” na stronie 94
W tym przykładzie przedstawiono sposób użycia funkcji API spawn() do tworzenia procesu potomnego, który dziedziczy deskryptor gniazda od procesu macierzystego.

Przykład: przekazywanie deskryptorów między procesami

W tych przykładach przedstawiono sposób projektowania programu serwerowego do obsługi połączeń przychodzących za pomocą funkcji API sendmsg() i recvmsg().

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.

Na poniższym rysunku przedstawiono współdziałanie serwera, procesu roboczego i klienta, gdy w systemie jest używany program serwera sendmsg() i recvmsg().



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

Poniżej wyjaśniono sekwencję wywołań funkcji API gniazd, przedstawionych na rysunku. Opisano także relacje między aplikacją serwera a przykładowym procesem roboczym. W pierwszym przykładzie wykorzystano następujące wywołania funkcji gniazd do tworzenia procesu potomnego za pomocą funkcji API sendmsg() i recvmsg():

1. Funkcja API socket() zwraca deskryptor gniazda reprezentujący punkt końcowy. Instrukcja ta informuje również, że dla tego gniazda zostanie użyta rodzina adresów INET (Internet Protocol) i protokół transportowy TCP (SOCK_STREAM).

2. Po utworzeniu deskryptora gniazda funkcja API bind() pobiera unikalną nazwę gniazda.
3. Funkcja API listen() pozwala serwerowi na przyjmowanie połączeń przychodzących od klientów.
4. Funkcja socketpair() tworzy parę datagramowych gniazd UNIX. Do utworzenia pary gniazd AF_UNIX serwer może również użyć funkcji socketpair().
5. Funkcja API 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 akceptuje żądanie połączenia przychodzącego za pomocą funkcji API accept(). Wywołanie funkcji API 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 za pomocą funkcji recvmsg(). Gdy serwer wywołuje funkcję sendmsg(), zadanie potomne nie jest aktywne.
8. W tym przykładzie pierwsze wywołanie funkcji close() zamyka gniazdo, które zaakceptowało połączenie. Drugie wywołanie funkcji close() zamyka gniazdo nasłuchujące.

Przebieg zdarzeń w gnieździe: zadanie procesu roboczego używające funkcji recvmsg()

W drugim przykładzie jest wykorzystywana następująca sekwencja wywołań funkcji API:

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

Odsyłacze pokrewne

“Przekazywanie deskryptorów między procesami - funkcje sendmsg() i recvmsg()” na stronie 73
Przekazanie otwartego deskryptora między zadaniami pozwala jednemu procesowi (zwykle procesowi serwera) wykonać wszystkie działania niezbędne do uzyskania deskryptora, w tym otworzyć plik, nawiązać połączenie i oczekiwać na zakończenie realizacji funkcji API accept(). Pozwala to także innemu procesowi (zwykle procesowi roboczemu) obsłużyć wszystkie operacje przesyłania danych, gdy tylko deskryptor zostanie otwarty.

“Zalecenia dotyczące projektowania aplikacji używających gniazd” na stronie 84
Przed przystąpieniem do pracy z aplikacją używającą gniazd należy ocenić jej wymagania funkcjonalne, cele i potrzeby. Należy również rozważyć wymagania aplikacji dotyczące wydajności oraz jej wpływ na zasoby systemu.

“Przykład: ogólny program klienta” na stronie 109
W przykładzie użyto kodu typowych zadań klienta. Zadanie klienta używa funkcji socket(), connect(), send(), recv() i close().

Informacje pokrewne

spawn()

Funkcja API bind() - ustawianie lokalnego adresu gniazda

Funkcja API socket() - tworzenie gniazd

Funkcja API listen() - nasłuchiwanie przychodzących żądań połączenia

Funkcja API accept() - oczekiwanie na żądanie i nawiązywanie połączenia

Funkcja API close() - zamykanie deskryptora gniazda lub pliku

Funkcja API socketpair() - tworzenie pary gniazd

Funkcja API sendmsg() - wysyłanie komunikatów przez gniazdo

Funkcja API recvmsg() - odbieranie komunikatów za pośrednictwem gniazda

Funkcja API send() - wysyłanie danych

Funkcja API recv() - odbieranie danych

Przykład: program serwera używany dla funkcji sendmsg() i recvmsg():

W tym przykładzie przedstawiono sposób użycia funkcji API sendmsg() do tworzenia puli zadań procesów roboczych.

Uwaga: Korzystając z przykładów, użytkownik wyraża zgodę na warunki podane w sekcji “Licencja na kod oraz Informacje dotyczące kodu” na stronie 199.

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

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

/*************************************************/
/* Ustaw parametr backlog funkcji listen.        */
/*************************************************/
rc = listen(listen_sd, 5);
if (rc < 0)
{
   perror("Niepowodzenie funkcji 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 funkcji socketpair()");
   close(listen_sd);
   exit(-1);
}
server_sd = pair_sd[0];
worker_sd = pair_sd[1];

/*************************************************/
/* Inicjuj parametry przed wejściem w pętlę for  */
/*                                               */
/* 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 funkcji 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++)
{
   /*********************************************/
   /* Czekaj na połączenie przychodzące         */
   /*********************************************/
   printf("Iteracja: %d\n", i+1);
   printf("  Oczekiwanie na wywołanie funkcji accept()\n");
   accept_sd = accept(listen_sd, NULL, NULL);
   if (accept_sd < 0)
   {
      perror("Niepowodzenie funkcji accept()");
      close(listen_sd);
      close(server_sd);
      exit(-1);
   }
   printf(" Wywołanie funkcji 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 funkcji 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         */
      /* procesowi roboczemu.                      */
      /*********************************************/
      close(accept_sd);
   }

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

Odsyłacze pokrewne

“Przykład: ogólny program klienta” na stronie 109
W przykładzie użyto kodu typowych zadań klienta. Zadanie klienta używa funkcji socket(), connect(), send(), recv() i close().

Przykład: program procesu roboczego używany dla funkcji sendmsg() i recvmsg():

W tym przykładzie przedstawiono sposób użycia zadania klienta funkcji API recvmsg() do odbierania zadań procesów roboczych.

Uwaga: Korzystając z przykładów, użytkownik wyraża zgodę na warunki podane w sekcji “Licencja na kod oraz Informacje dotyczące kodu” na stronie 199.

```
/*****************************************************************************/
/* 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;
```

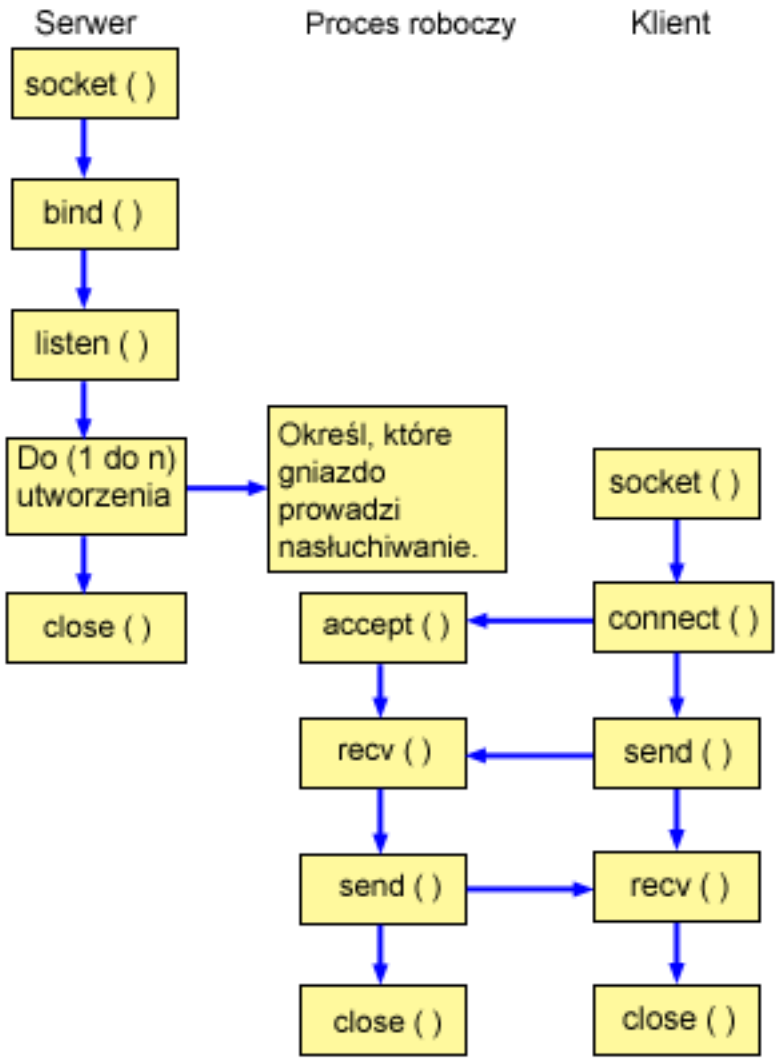
```
   msg.msg_iovlen  = 1;

   /************************************************/
   /* 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 funkcji 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);
   }

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

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

   /************************************************/
   /* Zamknij deskryptory                          */
   /************************************************/
   close(worker_sd);
   close(pass_sd);
}
```

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

W tych przykładach przedstawiono sposób projektowania programu serwerowego używającego modelu z wieloma funkcjami accept() do obsługi przychodzących żądań połączenia.

Gdy jest uruchamiany serwer używający wielu funkcji accept(), normalnie wywołuje on funkcje socket(), bind() i listen(). Następnie tworzy pulę zadań procesów roboczych i przypisuje każdemu zadaniu gniazdo nasłuchujące. Następnie każdy proces roboczy wielu funkcji accept() wywołuje funkcję accept().

Na poniższym rysunku przedstawiono współdziałanie serwera, procesu roboczego i klienta, gdy w systemie jest wykorzystywany program serwera używającego wielu funkcji accept().



Przebieg zdarzeń w gnieździe: serwer tworzący pulę wielu zadań procesów roboczych accept()

Poniżej wyjaśniono sekwencję wywołań funkcji API gniazd, przedstawionych na rysunku. Opisano także relacje między aplikacją serwera a przykładowym procesem roboczym. Każdy zbiór przepływów zawiera odsyłacze do uwag dotyczących użycia poszczególnych funkcji API. Więcej informacji na temat użycia tych funkcji API można uzyskać za pomocą wymienionych odsyłaczy. W pierwszym przykładzie wykorzystano następujące wywołania funkcji gniazd do tworzenia procesu potomnego:

1. Funkcja API socket() zwraca deskryptor gniazda reprezentujący punkt końcowy. Instrukcja ta informuje również, że dla tego gniazda zostanie użyta rodzina adresów INET (Internet Protocol) i protokół transportowy TCP (SOCK_STREAM).
2. Po utworzeniu deskryptora gniazda funkcja API bind() pobiera unikalną nazwę gniazda.
3. Funkcja API listen() pozwala serwerowi na przyjmowanie połączeń przychodzących od klientów.
4. Funkcja API spawn() tworzy poszczególne zadania procesu roboczego.
5. W tym przykładzie pierwsze wywołanie funkcji API close() zamyka gniazdo nasłuchujące.

Przebieg zdarzeń w gnieździe: zadanie procesu roboczego używające wielu funkcji accept()

W drugim przykładzie jest wykorzystywana następująca sekwencja wywołań funkcji API:

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

Odsyłacze pokrewne

“Przykład: ogólny program klienta” na stronie 109
W przykładzie użyto kodu typowych zadań klienta. Zadanie klienta używa funkcji socket(), connect(), send(), recv() i close().

Informacje pokrewne

spawn()

Funkcja API bind() - ustawianie lokalnego adresu gniazda

Funkcja API socket() - tworzenie gniazd

Funkcja API listen() - nasłuchiwanie przychodzących żądań połączenia

Funkcja API close() - zamykanie deskryptora gniazda lub pliku

Funkcja API accept() - oczekiwanie na żądanie i nawiązywanie połączenia

Funkcja API send() - wysyłanie danych

Funkcja API recv() - odbieranie danych

Przykład: program serwera tworzący pulę wielu procesów roboczych funkcji accept():

W tym przykładzie przedstawiono sposób użycia modeli wielu funkcji API accept() do tworzenia puli zadań procesów roboczych.

Uwaga: Korzystając z przykładów, użytkownik wyraża zgodę na warunki podane w sekcji “Licencja na kod oraz Informacje dotyczące kodu” na stronie 199.

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

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

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

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

```
   /* Zainicjuj parametry przed wejściem w pętlę for*/
   /*                                                */
   /* 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 funkcji 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);
}
```

Odsyłacze pokrewne

“Przykład: ogólny program klienta” na stronie 109
W przykładzie użyto kodu typowych zadań klienta. Zadanie klienta używa funkcji socket(), connect(), send(), recv() i close().

Przykład: procesy robocze dla wielu funkcji accept():

W tym przykładzie przedstawiono sposób odbierania zadań procesu roboczego i wywoływania serwera accept() przez wiele funkcji API accept().

Uwaga: Korzystając z przykładów, użytkownik wyraża zgodę na warunki podane w sekcji “Licencja na kod oraz Informacje dotyczące kodu” na stronie 199.

```
/*********************************************************************/
/* 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 jest przeka- */
   /* zywany do tego zadania procesu roboczego jako */
   /* parametr wiersza komend                       */
   /*************************************************/
   listen_sd = 0;

   /*************************************************/
   /* Czekaj na połączenie przychodzące             */
   /*************************************************/
   printf("  Oczekiwanie na wywołanie funkcji accept()\n");
   accept_sd = accept(listen_sd, NULL, NULL);
   if (accept_sd < 0)
   {
      perror("Niepowodzenie funkcji accept()");
      close(listen_sd);
      exit(-1);
   }
   printf(" Wywołanie funkcji Accept zakończone powodzeniem\n");

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

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

   /*************************************************/
   /* Zamknij deskryptory                           */
   /*************************************************/
   close(listen_sd);
   close(accept_sd);
}
```

Przykład: ogólny program klienta

W przykładzie użyto kodu typowych zadań klienta. Zadanie klienta używa funkcji socket(), connect(), send(), recv() i close().

Zadanie klienta nie ma informacji o tym, że wysłany bufor danych został odebrany i przekazany do zadania procesu roboczego, a nie do serwera. Aby utworzyć aplikację klienta, która działa niezależnie od tego, czy serwer używa rodziny adresów AF_INET, czy AF_INET6, należy zastosować przykład klienta IPv4 lub IPv6.

To zadanie klienta działa z każdą z następujących konfiguracji serwera zorientowanych na połączenie:

- Serwer iteracyjny - konfiguracja omówiona w przykładzie: Pisanie programu serwera iteracyjnego.
- Serwer potomny i proces roboczy - konfiguracja omówiona w przykładzie: Używanie funkcji spawn() do tworzenia procesów potomnych.
- Serwer sendmsg() i proces roboczy rcvmsg() - konfiguracja omówiona w przykładzie: Program serwerowy używany dla funkcji sendmsg() i recvmsg().
- Wielokrotna funkcja accept() - konfiguracja omówiona w przykładzie: Program serwerowy tworzący pulę wielu procesów roboczych funkcji accept().
- Nieblokujące operacje we/wy i funkcja select() - konfiguracja omówiona w przykładzie: Nieblokujące operacje we/wy i funkcja select().
- Serwer akceptujący połączenia z klienta IPv4 lub IPv6 - konfiguracja omówiona w przykładzie: Akceptowanie połączeń z klientów IPv6 i IPv4.

Przebieg zdarzeń w gnieździe: ogólny program klienta

Przedstawiony program używa następującej sekwencji wywołań funkcji API:

1. Funkcja API socket() zwraca deskryptor gniazda reprezentujący punkt końcowy. Instrukcja ta informuje również, że dla tego gniazda zostanie użyta rodzina adresów INET (Internet Protocol) i protokół transportowy TCP (SOCK_STREAM).
2. Po odebraniu deskryptora gniazda do nawiązania połączenia z serwerem należy użyć funkcji API connect().
3. Funkcja API send() odsyła bufory danych do procesów roboczych.
4. Funkcja API recv() odbiera bufory danych z procesów roboczych.
5. Funkcja API close() zamyka wszelkie otwarte deskryptory gniazd.

Uwaga: Korzystając z przykładów, użytkownik wyraża zgodę na warunki podane w sekcji “Licencja na kod oraz Informacje dotyczące kodu” na stronie 199.

```
/*********************************************************************/
/* Przykład ogólnego programu klienckiego 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 strumieniowe AF_INET          */
   /************************************************/
   sockfd = socket(AF_INET, SOCK_STREAM, 0);
   if (sockfd < 0)
   {
      perror("socket");
      exit(-1);
   }

   /************************************************/
   /* Zainicjuj strukturę 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);

   /*********************************************/
   /* Połącz się 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");

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

   /*********************************************/
   /* Wyślij bufor danych do zadania proc. roboczego*/
   /*********************************************/
   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);

   /*********************************************/
   /* Odbierz bufor danych z zadania proc. roboczego*/
   /*********************************************/
   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);
}
```

Odsyłacze pokrewne

“Przykład: klient IPv4 lub IPv6” na stronie 81
Tego programu przykładowego można użyć razem z aplikacją serwera, która akceptuje żądania od klientów IPv4 i IPv6.

“Przykłady: projekty aplikacji zorientowanych na połączenie” na stronie 87
System udostępnia kilka metod projektowania serwera używającego gniazd zorientowanego na połączenia. Do tworzenia własnych programów zorientowanych na połączenie można użyć poniższych programów przykładowych.

“Przykład: pisanie programu serwera iteracyjnego” na stronie 88
W tym przykładzie przedstawiono sposób tworzenia pojedynczego zadania serwera, które obsługuje wszystkie połączenia przychodzące. Po zakończeniu działania funkcji API accept() serwer obsługuje całą transakcję.

"Przykład: przekazywanie deskryptorów między procesami" na stronie 97
W tych przykładach przedstawiono sposób projektowania programu serwerowego do obsługi połączeń przychodzących za pomocą funkcji API sendmsg() i recvmsg().

"Przykład: program serwera używany dla funkcji sendmsg() i recvmsg()" na stronie 99
W tym przykładzie przedstawiono sposób użycia funkcji API sendmsg() do tworzenia puli zadań procesów roboczych.

"Przykłady: używanie wielu funkcji API accept() do obsługi żądań przychodzących" na stronie 105
W tych przykładach przedstawiono sposób projektowania programu serwerowego używającego modelu z wieloma funkcjami accept() do obsługi przychodzących żądań połączenia.

"Przykład: program serwera tworzący pulę wielu procesów roboczych funkcji accept()" na stronie 106
W tym przykładzie przedstawiono sposób użycia modeli wielu funkcji API accept() do tworzenia puli zadań procesów roboczych.

"Przykład: używanie funkcji API spawn() do tworzenia procesów potomnych" na stronie 92
W tym przykładzie przedstawiono sposób użycia przez program serwera funkcji API spawn() do utworzenia procesu potomnego, który dziedziczy deskryptor gniazda po procesie macierzystym.

"Przykład: akceptowanie połączeń od klientów IPv6 i IPv4" na stronie 76
W tym programie przykładowym przedstawiono sposób tworzenia modelu serwer/klient, w którym przyjmowane są żądania od aplikacji zarówno w standardzie IPv4 (aplikacje używające gniazd z rodziną adresów AF_INET), jak i IPv6 (aplikacje używające gniazd z rodziną adresów AF_INET6).

"Przykład: korzystanie z asynchronicznych operacji we/wy"
Aplikacja tworzy port zakończenia operacji we/wy za pomocą funkcji API QsoCreateIOCompletionPort(). Funkcja ta zwraca uchwyt, za pomocą którego można zaplanować asynchroniczne żądania we/wy i oczekiwać na ich zakończenie.

"Przykład: nieblokujące operacje we/wy i funkcja select()" na stronie 155
W tym przykładzie przedstawiono aplikację serwera, która wykorzystuje nieblokujące operacje we/wy i funkcję API select().

Informacje pokrewne

Funkcja API socket() - tworzenie gniazd

Funkcja API connect() - nawiązywanie połączenia lub ustanawianie adresu docelowego

Funkcja API close() - zamykanie deskryptora gniazda lub pliku

Funkcja API send() - wysyłanie danych

Funkcja API recv() - odbieranie danych

Przykład: korzystanie z asynchronicznych operacji we/wy

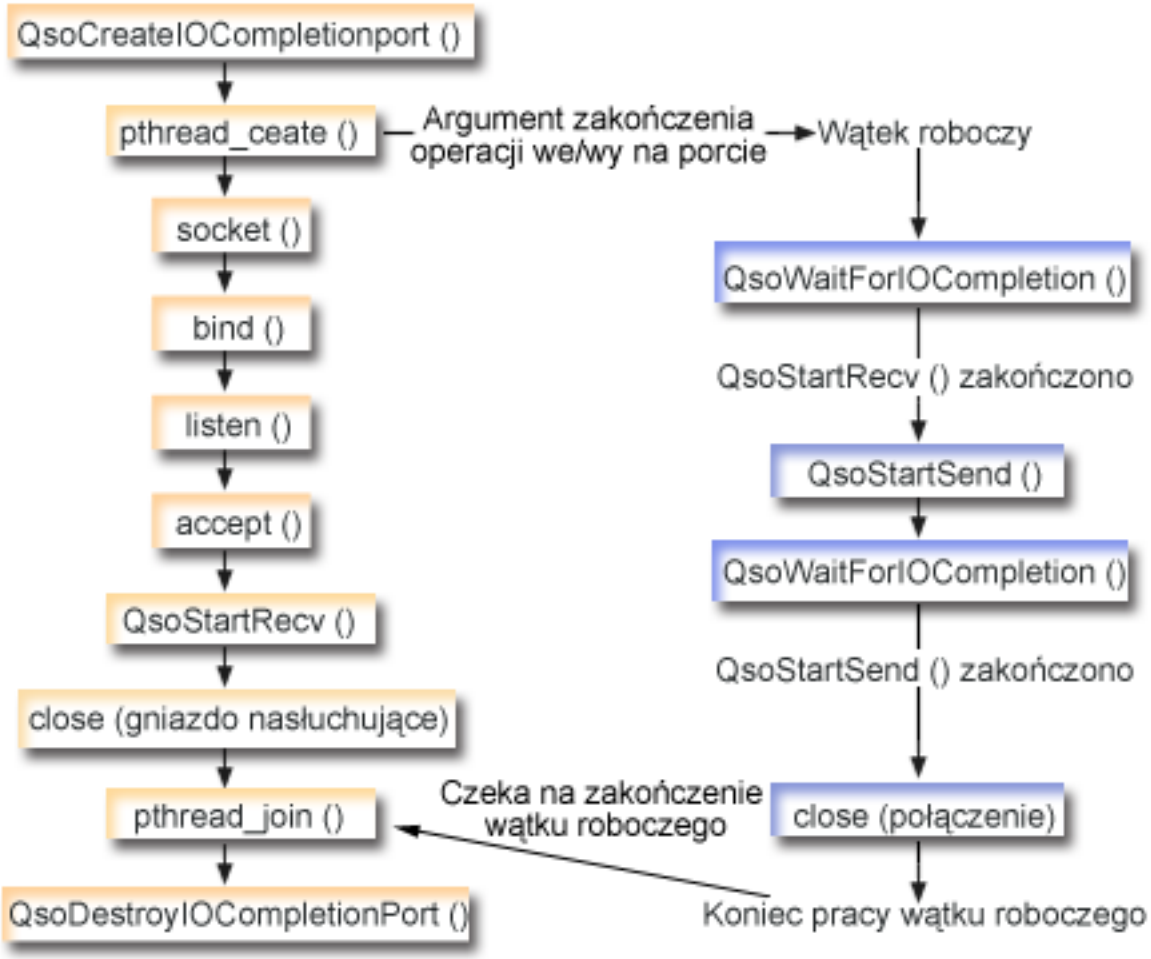
Aplikacja tworzy port zakończenia operacji we/wy za pomocą funkcji API QsoCreateIOCompletionPort(). Funkcja ta zwraca uchwyt, za pomocą którego można zaplanować asynchroniczne żądania we/wy i oczekiwać na ich zakończenie.

Następnie aplikacja uruchamia funkcję operacji wejścia lub operacji wyjścia przez podanie uchwytu portu zakończenia operacji we/wy. Kiedy operacje we/wy zostają zakończone, informacje o statusie i uchwyt zdefiniowany przez aplikację zostają zapisane w podanym porcie zakończenia operacji we/wy. Zapisanie do portu zakończenia operacji we/wy uaktywnia dokładnie jeden z wielu możliwych wątków oczekujących. Aplikacja odbiera następujące dane:

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

Ten uchwyt aplikacji może być deskryptorem gniazda, wskazującym połączenie klienta, lub wskaźnikiem pamięci, która zawiera dodatkowe informacje o stanie połączenia klienta. Ponieważ operacja się zakończyła, 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 obsłużyć wiele różnych żądań

klienta i nie są powiązane z tylko jednym z nich. Ponieważ kopiowanie z i do buforów użytkowników odbywa się asynchronicznie względem procesów serwera, czasy oczekiwania dla żądań klientów są krótsze. Cecha ta może być szczególnie korzystna w systemach wieloprocesorowych.



Przebieg zdarzeń w gnieździe: asynchroniczny serwer we/wy

Poniżej wyjaśniono sekwencję wywołań funkcji API gniazd, przedstawionych na rysunku. Opisano także relacje między aplikacją serwera a przykładowym procesem roboczym. Każdy zbiór przepływów zawiera odsyłacze do uwag dotyczących użycia poszczególnych funkcji API. Więcej informacji na temat użycia tych funkcji API można uzyskać za pomocą wymienionych odsyłaczy. Ten przepływ przedstawia wywołania funkcji gniazd w poniższej aplikacji przykładowej. Serwer ten może współpracować z przykładowym klientem ogólnym.

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

   Uwaga: Akceptacja może się również odbyć w trybie asynchronicznym z użyciem funkcji QsoStartAccept().

5. W pewnym momencie żądanie klienta w sposób asynchroniczny dociera do procesu serwera. System operacyjny gniazd wczytuje dostarczony bufor użytkownika i wysyła zakończone żądanie wywołania funkcji QsoStartRecv() do określonego portu zakończenia operacji we/wy. Zostaje uaktywniony jeden wątek procesu roboczego, który kontynuuje przetwarzanie tego żądania.
6. Wątek procesu roboczego wyodrębnia deskryptor gniazda z uchwytu zdefiniowanego przez aplikację i odsyła odebrane dane do klienta za pomocą funkcji QsoStartSend().
7. Jeśli dane mogą być wysłane natychmiast, to funkcja QsoStartSend() zwróci odpowiednią informację; w przeciwnym razie system operacyjny gniazd prześle te dane możliwie szybko i zapisze informację o tym fakcie w określonym porcie zakończenia operacji we/wy. Wątek procesu roboczego pobiera informację o wysłanych danych i oczekuje w porcie zakończenia operacji we/wy na kolejne żądanie lub zostaje zakończony, jeśli wystąpi taka instrukcja. Wątek główny może zapisać zdarzenie zakończenia wątku procesu roboczego za pomocą funkcji QsoPostIOCompletion().
8. Wątek główny oczekuje na zakończenie zadania przez wątek procesu roboczego, a następnie niszczy port zakończenia operacji we/wy przez wywołanie funkcji QsoDestroyIOCompletionPort().

Uwaga: Korzystając z przykładów, użytkownik wyraża zgodę na warunki podane w sekcji “Licencja na kod oraz Informacje dotyczące kodu” na stronie 199.

```
      #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: Ze względu 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;


       /*******************************************/
       /* Utwórz port zakończenia operacji we/wy    */
       /* dla tego procesu.                       */
       /*******************************************/
       if ((ioCompPort = QsoCreateIOCompletionPort()) < 0)
       {
```

```
  perror("Niepowodzenie funkcji 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 zakończ. operacji we/wy */
/*******************************************/
rc = pthread_create(&thr, NULL, workerThread,
                    &ioCompPort);
if (rc < 0)
{
  perror("Niepowodzenie funkcji pthread_create()");
  QsoDestroyIOCompletionPort(ioCompPort);
  close(listen_sd);
  exit(-1);
}

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

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

/*******************************************/
/* Ustawia kolejkę (backlog) nasłuchiwania.  */
/*******************************************/
```

```
if ((rc = listen(listen_sd, 10)) < 0)
{
  perror("Niepowodzenie funkcji 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 funkcji accept()");
  QsoDestroyIOCompletionPort(ioCompPort);
  close(listen_sd);
  exit(-1);
}

/*********************************************/
/* Wywołuje QsoStartRecv(), aby odebrać      */
/*             żądanie klienta.              */
/* Uwaga:                                    */
/* postFlag == podczas odczytywania żądania  */
/*             należy je przesłać do portu   */
/*             zakończenia operacji we/wy,   */
/*             nawet jeśli 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 funkcji QsoStartRecv()");
  QsoDestroyIOCompletionPort(ioCompPort);
  close(listen_sd);
  close(client_sd);
```

```
        exit(-1);
      }
      /*******************************************/
      /* Zamknięcie gniazda nasłuchującego serwera */
      /*******************************************/
      close(listen_sd);

      /*******************************************/
      /* Czekaj, aż wątek procesu roboczego zakoń- */
      /* czy 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 funkcji pthread_join()");
        exit(-1);
      }
}
/* koniec 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 zakończenia operacji we/wy jest      */
      /* przekazywany do tej procedury.            */
      /*******************************************/
      ioCompPort = *(int *)arg;

      /*******************************************/
      /* Czekaj przy dostarczonym porcie zakończe- */
      /* nia operacji we/wy 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 zakończenia operacji we/wy    */
    /* 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 zakończenia      */
    /* operacji we/wy 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 się powiodło.                          */
     /*******************************************/
        ;
      else
      {
        printf("Niepowodzenie QsoWaitForIOCompletion() lub QsoStartSend().\n");
        perror("Niepowodzenie QsoWaitForIOCompletion() lub QsoStartSend()");
        return __VOID(Failure);
      }
    }
    close(clientfd);
    return __VOID(Success);
} /* koniec workerThread */
```

Pojęcia pokrewne

"Asynchroniczne operacje we/wy" na stronie 42
Funkcje API asynchronicznych operacji we/wy udostępniają metodę realizacji wątkowych modeli klient/serwer w celu zapewnienia wysoko współbieżnych operacji we/wy z efektywnym wykorzystaniem pamięci.

Odsyłacze pokrewne

"Zalecenia dotyczące projektowania aplikacji używających gniazd" na stronie 84
Przed przystąpieniem do pracy z aplikacją używającą gniazd należy ocenić jej wymagania funkcjonalne, cele i potrzeby. Należy również rozważyć wymagania aplikacji dotyczące wydajności oraz jej wpływ na zasoby systemu.

"Przykłady: projekty aplikacji zorientowanych na połączenie" na stronie 87
System udostępnia kilka metod projektowania serwera używającego gniazd zorientowanego na połączenia. Do tworzenia własnych programów zorientowanych na połączenie można użyć poniższych programów przykładowych.

"Przykład: ogólny program klienta" na stronie 109
W przykładzie użyto kodu typowych zadań klienta. Zadanie klienta używa funkcji socket(), connect(), send(), recv() i close().

"Przykład: używanie sygnałów z blokującymi funkcjami API gniazd" na stronie 168
Sygnały mogą przekazywać powiadomienie o tym, że doszło do zablokowania procesu lub aplikacji. Udostępniają także limit czasu blokowania procesów.

Informacje pokrewne

Funkcja API QsoCreateIOCompletionPort() - tworzenie portu zakończenia operacji we/wy

pthread_create

Funkcja API QsoWaitForIOCompletion() - oczekiwanie na operację we/wy

Funkcja API QsoStartAccept() - uruchamianie asynchronicznej operacji akceptowania

Funkcja API QsoStartSend() - uruchamianie asynchronicznej operacji wysyłania

Funkcja API QsoDestroyIOCompletionPort() - likwidacja portu zakończenia operacji we/wy

Przykłady: nawiązywanie połączeń chronionych

Do utworzenia chronionego serwera i klienta można użyć funkcji API GSKit lub funkcji SSL_.

Preferowane są funkcje API GSKit, ponieważ są one obsługiwane we wszystkich systemach IBM, podczas gdy funkcje API SSL_ występują tylko w systemie operacyjnym i5/OS. Obydwa zestawy funkcji API gniazd chronionych mają kody powrotu, które pomagają zidentyfikować błędy powstałe podczas nawiązywania połączeń chronionych.

Uwaga: Korzystając z przykładów, użytkownik wyraża zgodę na warunki podane w sekcji "Licencja na kod oraz Informacje dotyczące kodu" na stronie 199.

Pojęcia pokrewne

"Komunikaty z kodami błędów funkcji API gniazd chronionych" na stronie 50
Aby odczytać komunikaty zawierające kody błędów funkcji API związanych z obsługą gniazd chronionych, należy wykonać następujące czynności.

Przykład: chroniony serwer GSKit z asynchronicznym odbieraniem danych

W tym przykładzie przedstawiono sposób ustanawiania chronionego serwera za pomocą funkcji API 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 między serwerem a klientem.

Uwaga: W poniższych przykładach jest używana rodzina adresów AF_INET, ale można je zmodyfikować tak, aby była używana również rodzina adresów AF_INET6.

Przebieg zdarzeń w gnieździe: chroniony serwer z asynchronicznym odbieraniem danych

Poniżej wyjaśniono sekwencję wywołań funkcji API gniazd, przedstawionych na rysunku. Opisano także relacje między aplikacją serwera a aplikacją klienta.

1. Funkcja API QsoCreateIOCompletionPort() tworzy port zakończenia operacji we/wy.
2. Funkcja API pthread_create tworzy wątek procesu roboczego, który będzie odbierał dane i odsyłał je do klienta. Wątek procesu roboczego oczekuje na nadejście żądań od klienta do właśnie utworzonego portu zakończenia operacji we/wy.
3. Wywołanie funkcji API gsk_environment_open() w celu uzyskania uchwytu środowiska SSL.
4. Co najmniej jedno wywołanie funkcji API gsk_attribute_set_xxxxx() w celu ustawienia atrybutów środowiska SSL. Minimum to wywołanie funkcji API 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. Należy ponadto ustawić typ aplikacji (klient lub serwer) i wartość GSK_SESSION_TYPE przy użyciu funkcji API gsk_attribute_set_enum().
5. Wywołanie funkcji API gsk_environment_init() w celu zainicjowania tego środowiska do przetwarzania SSL i określenia informacji o bezpieczeństwie dla wszystkich sesji SSL, które będą uruchamiane w tym środowisku.
6. Funkcja API socket tworzy deskryptor gniazda. Serwer wywołuje następnie standardowy zestaw funkcji gniazd: bind(), listen() i accept() w celu umożliwienia akceptowania przychodzących żądań połączenia.
7. Funkcja API gsk_secure_soc_open() uzyskuje pamięć dla sesji chronionej, ustawia domyślne wartości atrybutów oraz zwraca uchwyt, który musi zostać zapisany i użyty podczas kolejnych wywołań funkcji API związanych z sesją chronioną.
8. Co najmniej jedno wywołanie funkcji API gsk_attribute_set_xxxxx() w celu ustawienia atrybutów sesji chronionej. Minimum to wywołanie funkcji API gsk_attribute_set_numeric_value() w celu powiązania określonego gniazda z tą sesją chronioną.
9. Wywołanie funkcji gsk_secure_soc_init() w celu zainicjowania negocjacji parametrów szyfrowania podczas uzgadniania SSL.

   Uwaga: Zwykle aby uzgadnianie SSL się powiodło, program serwera musi okazać certyfikat. Serwer musi ponadto mieć dostęp do klucza prywatnego powiązanego z certyfikatem serwera i do zbioru bazy danych kluczy, w którym jest przechowywany certyfikat. W niektórych przypadkach także klient musi okazać certyfikat podczas przetwarzania uzgadniania SSL. Dotyczy to sytuacji, gdy na serwerze, z którym łączy się klient, włączono uwierzytelnianie klienta. Wywołania funkcji API gsk_attribute_set_buffer (GSK_OS400_APPLICATION_ID) lub gsk_attribute_set_buffer (GSK_KEYRING_FILE) identyfikują (w różny sposób) zbiór bazy danych kluczy, za pomocą którego uzyskano certyfikat i klucz prywatny użyte podczas uzgadniania.
10. Funkcja API gsk_secure_soc_startRecv() inicjuje asynchroniczną operację odbierania w ramach sesji chronionej.
11. Funkcja API 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 API gsk_secure_soc_close() kończy sesję chronioną.
13. Funkcja API gsk_environment_close() zamyka środowisko SSL.
14. Funkcja API close() zamyka gniazdo nasłuchujące.
15. Funkcja close() zamyka gniazdo, które zaakceptowało połączenie klienta.
16. Funkcja API QsoDestroyIOCompletionPort() niszczy port zakończenia.

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

1. Wątek procesu roboczego utworzony przez aplikację serwera oczekuje na wysłanie przez serwer przychodzącego żądania klienta w celu obsłużenia danych klienta za pomocą wywołania funkcji API gsk_secure_soc_startRecv(). Funkcja API QsoWaitForIOCompletionPort() oczekuje na dostarczonym porcie zakończenia operacji we/wy określonym przez serwer.
2. Po otrzymaniu żądania klienta funkcja API gsk_attribute_get_numeric_value() pobiera deskryptor gniazda powiązany z sesją chronioną.
3. Funkcja API gsk_secure_soc_write() wysyła komunikat do klienta w ramach sesji chronionej.

Uwaga: Korzystając z przykładów, użytkownik wyraża zgodę na warunki podane w sekcji “Licencja na kod oraz Informacje dotyczące kodu” na stronie 199.

```
/* Program asynchronicznego serwera GSK używający ID 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            */
/* zarejestrowany i powiązany z certyfikatem.         */
/*                                                    */
/* Brak parametrów, nieco 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: Ze względu 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
  {
    /*******************************************/
    /* Utwórz port zakończenia operacji we/wy    */
    /* dla tego procesu.                         */
    /*******************************************/
    if ((ioCompPort = QsoCreateIOCompletionPort()) < 0)
    {
      perror("Niepowodzenie funkcji 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 zakończ. operacji we/wy */
    /*******************************************/
    rc = pthread_create(&thr, NULL, workerThread, &ioCompPort);
    if (rc < 0)
    {
      perror("Niepowodzenie funkcji pthread_create()");
      break;
    }

    /* otwórz 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 i kodem 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 i kodem 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 i kodem 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 zostanie wyłączony 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 i kodem błędu = %d.\n",
         rc,errno);
  printf("kod powrotu %d oznacza %s\n", rc, gsk_strerror(rc));
  break;
}

/* Określ, jakiego zestawu algorytmów szyfrowania użyć. Domyślnie jest */
/* włącz. domyślna lista szyfrowania. W tym przykł. zostanie ona użyta */
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 i kodem błędu = %d.\n"
         ,rc,errno);
  printf("kod powrotu %d oznacza %s\n", rc, gsk_strerror(rc));
  break;
}

/* Zainicjuj środowisko chronione       */
rc = errno = 0;
rc = gsk_environment_init(my_env_handle);
if (rc != GSK_OK)
{
  printf("Niepowodzenie gsk_environment_init() z kodem powrotu = %d i kodem 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 funkcji socket()");
  break;
}

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

```
    /* powiązanie z adresem 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 funkcji bind()");
      break;
    }

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

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

    /* otwórz sesję chronioną                   */
    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 i kodem błędu = %d.\n",
             rc,errno);
      printf("kod powrotu %d oznacza %s\n", rc, gsk_strerror(rc));
      break;
    }

    /* powiąż gniazdo 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 uzgadniania 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 i kodem błędu = %d.\n",
             rc,errno);
      printf("kod powrotu %d oznacza %s\n", rc, gsk_strerror(rc));
      break;
    }
/*******************************************/
    /* Wywołaj funkcję gsk_secure_soc_startRecv()*/
    /* do odebrania żądania klienta.             */
    /* Uwaga:                                    */
```

```
  /* postFlag == podczas odczytywania żądania  */
  /*             należy je przesłać do portu   */
  /*             zakończenia operacji, 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);

  /*********************************************/
  /* Przechowuj 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, kod 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.*/
  /*********************************************/

  /*********************************************/
  /* Czekaj, 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 funkcji pthread_join()");
  }
} while(FALSE);

/* wyłącz sesję SSL */
if (my_session_handle != NULL)
  gsk_secure_soc_close(&my_session_handle);
```

```
  /* wyłącz środowisko 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            */
/*        wartości 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 zakończenia operacji we/wy jest      */
    /* przekazywany do tej procedury.            */
    /*********************************************/
    ioCompPort = *(int *)arg;
    /*********************************************/
    /* Czekaj przy dostarczonym porcie zakończe- */
    /* nia operacji we/wy 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);
    }
```

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

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

    /* powiąż gniazdo 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);
    }

    /* wyślij komunikat do klienta w ramach sesji chronionej */
    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);
      }
    }

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

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

Pojęcia pokrewne

“Funkcje API z zestawu Global Secure ToolKit (GSKit)” na stronie 46
GSKit to zestaw programowalnych interfejsów, który umożliwia aplikacjom obsługę warstwy SSL.

Odsyłacze pokrewne

“Przykład: ustanawianie chronionego klienta za pomocą funkcji API Global Secure ToolKit” na stronie 139
W tym przykładzie przedstawiono sposób ustanawiania klienta za pomocą funkcji API GSKit.

“Przykład: chroniony serwer GSKit z uzgadnianiem asynchronicznym” na stronie 129
Funkcja API gsk_secure_soc_startInit() umożliwia tworzenie chronionych aplikacji serwera, które obsługują żądania w sposób asynchroniczny.

Informacje pokrewne

Funkcja API QsoCreateIOCompletionPort() - tworzenie portu zakończenia operacji we/wy

pthread_create

Funkcja API QsoWaitForIOCompletion() - oczekiwanie na operację we/wy

Funkcja API QsoDestroyIOCompletionPort() - likwidacja portu zakończenia operacji we/wy

Funkcja API bind() - ustawianie lokalnego adresu gniazda

Funkcja API socket() - tworzenie gniazd

Funkcja API listen() - nasłuchiwanie przychodzących żądań połączenia

Funkcja API close() - zamykanie deskryptora gniazda lub pliku

Funkcja API accept() - oczekiwanie na żądanie i nawiązywanie połączenia

Funkcja API gsk_environment_open() - pobierania uchwytu dla środowiska SSL

Funkcja API gsk_attribute_set_buffer() - ustawianie informacji znakowych dla sesji chronionej lub środowiska SSL

Funkcja API gsk_attribute_set_enum() - ustawianie informacji wyliczeniowych dla sesji chronionej lub środowiska SSL

Funkcja API gsk_environment_init() - inicjowanie środowiska SSL

Funkcja API gsk_secure_soc_open() - pobieranie uchwytu dla sesji chronionej

Funkcja API gsk_attribute_set_numeric_value() - ustawianie informacji liczbowych dla sesji chronionej lub środowiska SSL

Funkcja API gsk_secure_soc_init() - uzgadnianie sesji chronionej

Funkcja API gsk_secure_soc_startRecv() - uruchamianie asynchronicznej operacji odbierania podczas sesji chronionej

pthread_join

Funkcja API gsk_secure_soc_close() - zamykanie sesji chronionej

Funkcja API gsk_environment_close() - zamykanie środowiska SSL

Funkcja API gsk_attribute_get_numeric_value() - uzyskiwanie informacji liczbowych o sesji chronionej lub środowisku SSL

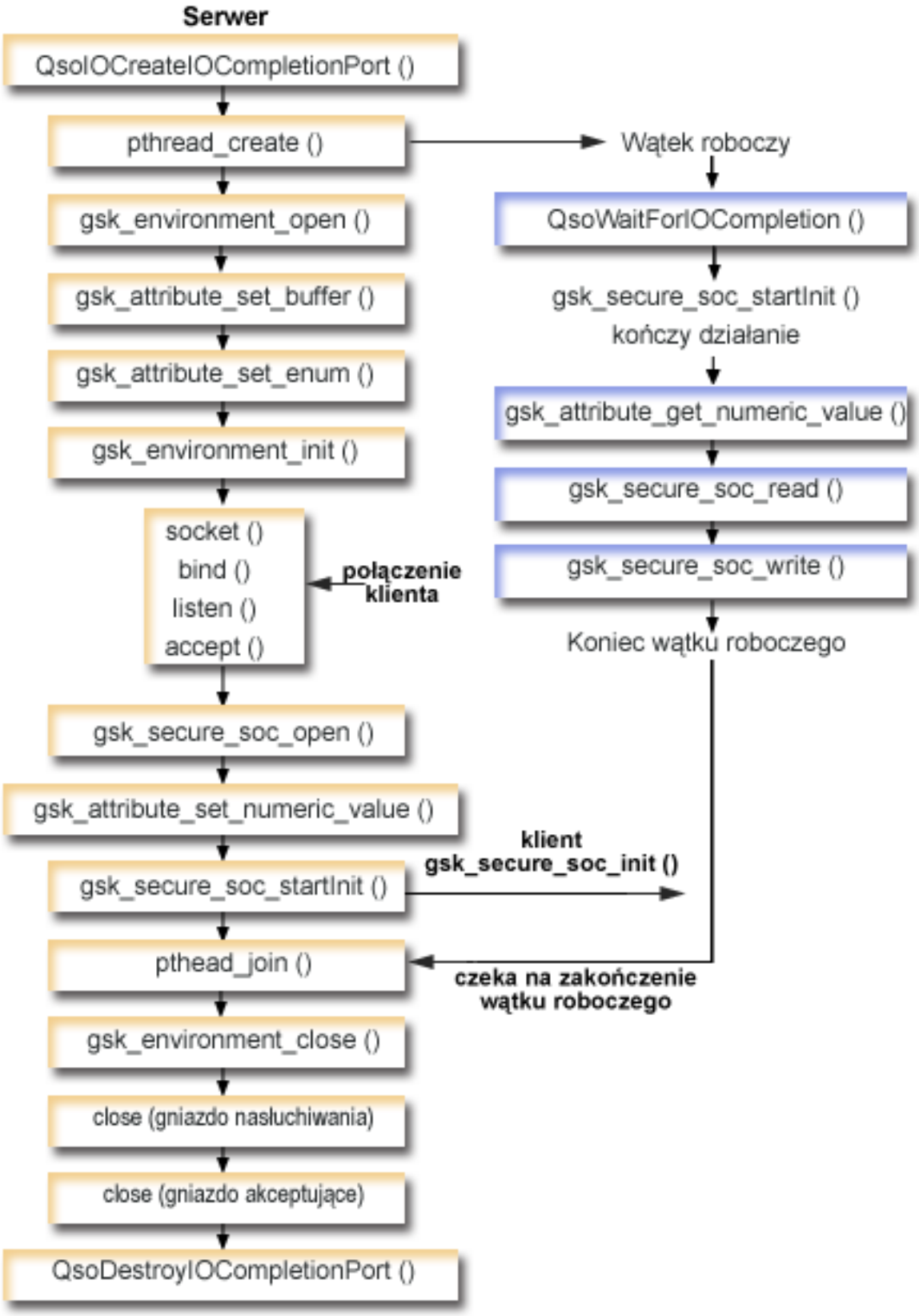
Funkcja API gsk_secure_soc_write() - wysyłanie danych podczas sesji chronionej

Przykład: chroniony serwer GSKit z uzgadnianiem asynchronicznym

Funkcja API gsk_secure_soc_startInit() umożliwia tworzenie chronionych aplikacji serwera, które obsługują żądania w sposób asynchroniczny.

W tym przykładzie przedstawiono sposób wykorzystania tej funkcji API. Przykład ten jest podobny do przykładu chronionego serwera GSKit z asynchronicznym odbieraniem danych, ale do rozpoczęcia chronionej sesji użyto opisanej tu funkcji API.

Na poniższym rysunku przedstawiono wywołania funkcji API służące do negocjowania uzgadniania asynchronicznego na 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 nasluchiwania)
close (gniazdo akceptujące)
QsoDestroyIOCompletionPort ()

Część rysunku dotycząca klienta znajduje się w sekcji poświęconej klientowi GSKit.

Przebieg zdarzeń w gnieździe: chroniony serwer GSKit z uzgadnianiem asynchronicznym

Ten przepływ przedstawia wywołania funkcji gniazd w poniższej aplikacji przykładowej.

1. Funkcja API QsoCreateIOCompletionPort() tworzy port zakończenia operacji we/wy.
2. Funkcja API pthread_create() tworzy wątek procesu roboczego, który będzie przetwarzał wszystkie żądania klienta. Wątek procesu roboczego oczekuje na nadejście żądań od klienta do właśnie utworzonego portu zakończenia operacji we/wy.
3. Wywołanie funkcji API gsk_environment_open() w celu uzyskania uchwytu środowiska SSL.
4. Co najmniej jedno wywołanie funkcji API gsk_attribute_set_xxxxx() w celu ustawienia atrybutów środowiska SSL. Minimum to wywołanie funkcji API 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. Należy ponadto ustawić typ aplikacji (klient lub serwer) i wartość GSK_SESSION_TYPE przy użyciu funkcji API gsk_attribute_set_enum().
5. Wywołanie funkcji API gsk_environment_init() w celu zainicjowania tego środowiska do przetwarzania SSL i określenia informacji o bezpieczeństwie dla wszystkich sesji SSL, które będą uruchamiane w tym środowisku.
6. Funkcja API socket tworzy deskryptor gniazda. Serwer wywołuje następnie standardowy zestaw funkcji gniazd: bind(), listen() oraz accept(), aby możliwe było akceptowanie przychodzących żądań połączenia.
7. Funkcja API gsk_secure_soc_open() uzyskuje pamięć dla sesji chronionej, ustawia domyślne wartości atrybutów oraz zwraca uchwyt, który musi zostać zapisany i użyty podczas kolejnych wywołań funkcji API związanych z sesją chronioną.
8. Co najmniej jedno wywołanie funkcji API gsk_attribute_set_xxxxx() w celu ustawienia atrybutów sesji chronionej. Minimum to wywołanie funkcji API gsk_attribute_set_numeric_value() w celu powiązania określonego gniazda z tą sesją chronioną.
9. Funkcja API gsk_secure_soc_startInit() uruchamia asynchroniczne uzgadnianie sesji chronionej przy użyciu zestawu atrybutów dla środowiska SSL i sesji chronionej. W tym miejscu sterowanie jest z powrotem przekazywane do programu. Po zakończeniu procesu uzgadniania port zakończenia zostaje zaktualizowany razem z rezultatami. Wątek może kontynuować przetwarzanie; jednak dla uproszczenia program oczekuje na zakończenie pracy wątku procesu roboczego.

   Uwaga: Zwykle aby uzgadnianie SSL się powiodło, program serwera musi okazać certyfikat. Serwer musi ponadto mieć dostęp do klucza prywatnego powiązanego z certyfikatem serwera i do zbioru bazy danych kluczy, w którym jest przechowywany certyfikat. W niektórych przypadkach także klient musi okazać certyfikat podczas przetwarzania uzgadniania SSL. Dotyczy to sytuacji, gdy na serwerze, z którym łączy się klient, włączono uwierzytelnianie klienta. Wywołanie funkcji API gsk_attribute_set_buffer (GSK_OS400_APPLICATION_ID) lub gsk_attribute_set_buffer (GSK_KEYRING_FILE) identyfikuje (w różny sposób) zbiór bazy danych kluczy, za pomocą którego uzyskano certyfikat i klucz prywatny użyte podczas uzgadniania.
10. Funkcja API 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 API gsk_secure_soc_close() kończy sesję chronioną.
12. Funkcja API gsk_environment_close() zamyka środowisko SSL.
13. Funkcja API close() zamyka gniazdo nasłuchujące.
14. Funkcja API close() zamyka gniazdo, które zaakceptowało połączenie klienta.
15. Funkcja API QsoDestroyIOCompletionPort() niszczy port zakończenia.

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

1. Wątek procesu roboczego utworzony przez aplikację serwera oczekuje na wysłanie przez serwer przychodzącego żądania klienta w celu przetworzenia tego żądania. Funkcja API QsoWaitForIOCompletionPort() oczekuje na dostarczony port zakończenia operacji we/wy przekazany jej przez serwer. Wywołanie to oczekuje na zakończenie działania funkcji API gsk_secure_soc_startInit().
2. Po otrzymaniu żądania klienta funkcja API gsk_attribute_get_numeric_value() pobiera deskryptor gniazda powiązany z sesją chronioną.
3. Funkcja API gsk_secure_soc_read() odbiera komunikat od klienta w ramach sesji chronionej.
4. Funkcja API gsk_secure_soc_write() wysyła komunikat do klienta w ramach sesji chronionej.

Uwaga: Korzystając z przykładów, użytkownik wyraża zgodę na warunki podane w sekcji “Licencja na kod oraz Informacje dotyczące kodu” na stronie 199.

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

/* Przyjmuje się, że ID aplikacji jest już                     */
/* zarejestrowany i powiązany z certyfikatem.                  */
/*                                                             */
/* Brak parametrów, nieco 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: Ze względu 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
{
  /*******************************************/
  /* Utwórz port zakończenia operacji we/wy    */
  /* dla tego procesu.                         */
  /*******************************************/
  if ((ioCompPort = QsoCreateIOCompletionPort()) < 0)
  {
    perror("Niepowodzenie funkcji 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 zakończ. operacji we/wy */
  /*******************************************/
  rc = pthread_create(&thr, NULL, workerThread, &ioCompPort);
  if (rc < 0)
  {
    perror("Niepowodzenie funkcji 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 i kodem 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 funcji gsk_attribute_set_buffer() z kodem powrotu = %d i kodem 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 i kodem 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 zostanie wyłączony 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 i kodem błędu = %d.\n",
            rc,errno);
      printf("kod powrotu %d oznacza %s\n", rc, gsk_strerror(rc));
      break;
    }

    /* Określ, jakiego zestawu algorytmów szyfrowania użyć. Domyślnie jest */
    /* włącz. domyślna lista szyfrowania. W tym przykł. zostanie ona użyta */
    rc = errno = 0;
    rc = gsk_attribute_set_buffer(my_env_handle,
                                  GSK_V3_CIPHER_SPECS,
                                  "05", 7* SSL_RSA_WITH_RC4_128_SHA */
                                  2);
    if (rc != GSK_OK)
    {
      printf("Niepowodzenie funcji gsk_attribute_set_buffer() z kodem powrotu = %d i kodem błędu = %d.\n"
            ,rc,errno);
      printf("kod powrotu %d oznacza %s\n", rc, gsk_strerror(rc));
      break;
    }

    /* Zainicjuj środowisko chronione   */
    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 i kodem 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 funkcji socket()");
      break;
    }

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

    /* powiązanie z adresem 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 funkcji bind()");
  break;
}

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

/* akceptacja przychodzącego połączenia klienta */
al = sizeof(address);
printf("accept()\n");
sd = accept(lsd, (struct sockaddr *) &address, &al);
if (sd < 0)
{
  perror("Niepowodzenie funkcji 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_soc_open() z kodem powrotu = %d i kodem błędu = %d.\n",
         rc,errno);
  printf("kod powrotu %d oznacza %s\n", rc, gsk_strerror(rc));
  break;
}
/* powiąż gniazdo 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));
```

```
  /*******************************************/
  /* Przechowuj 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 uzgadniania 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.*/
  /*******************************************/

  /*******************************************/
  /* Czekaj, 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 funkcji 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);

/* zamknij gniazdo nasłuchujące */
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            */
/*        wartości 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 zakończenia operacji we/wy jest      */
    /* przekazywany do tej procedury.            */
    /*********************************************/
    ioCompPort = *(int *)arg;
    /*********************************************/
    /* Czekaj przy dostarczonym porcie zakończe- */
    /* nia operacji 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);
    }

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

```
    /* powiąż gniazdo 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;
    /* odbierz komunikat od klienta w ramach sesji chronionej */
    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;
    }

    /* wyświetl wyniki na ekranie */
    printf("Funkcja gsk_secure_soc_read() odebrała bajtów: %d, oto one:\n",
         amtRead);
    printf("%s\n",buff);


    /* wyślij komunikat do klienta w ramach sesji chronionej */
    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);
      }
    }
    /* wyświetl wyniki na ekranie */
    printf("Funkcja gsk_secure_soc_write() zapisała bajtów: %d ...\n", amtWritten);
    printf("%s\n",buff);

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

Pojęcia pokrewne

“Funkcje API z zestawu Global Secure ToolKit (GSKit)” na stronie 46
GSKit to zestaw programowalnych interfejsów, który umożliwia aplikacjom obsługę warstwy SSL.

Odsyłacze pokrewne

“Przykład: ustanawianie chronionego klienta za pomocą funkcji API Global Secure ToolKit”
W tym przykładzie przedstawiono sposób ustanawiania klienta za pomocą funkcji API GSKit.

“Przykład: chroniony serwer GSKit z asynchronicznym odbieraniem danych” na stronie 119
W tym przykładzie przedstawiono sposób ustanawiania chronionego serwera za pomocą funkcji API GSKit.

Informacje pokrewne

Funkcja API QsoCreateIOCompletionPort() - tworzenie portu zakończenia operacji we/wy

pthread_create

Funkcja API QsoWaitForIOCompletion() - oczekiwanie na operację we/wy

Funkcja API QsoDestroyIOCompletionPort() - likwidacja portu zakończenia operacji we/wy

Funkcja API bind() - ustawianie lokalnego adresu gniazda

Funkcja API socket() - tworzenie gniazd

Funkcja API listen() - nasłuchiwanie przychodzących żądań połączenia

Funkcja API close() - zamykanie deskryptora gniazda lub pliku

Funkcja API accept() - oczekiwanie na żądanie i nawiązywanie połączenia

Funkcja API gsk_environment_open() - pobierania uchwytu dla środowiska SSL

Funkcja API gsk_attribute_set_buffer() - ustawianie informacji znakowych dla sesji chronionej lub środowiska SSL

Funkcja API gsk_attribute_set_enum() - ustawianie informacji wyliczeniowych dla sesji chronionej lub środowiska SSL

Funkcja API gsk_environment_init() - inicjowanie środowiska SSL

Funkcja API gsk_secure_soc_open() - pobieranie uchwytu dla sesji chronionej

Funkcja API gsk_attribute_set_numeric_value() - ustawianie informacji liczbowych dla sesji chronionej lub środowiska SSL

Funkcja API gsk_secure_soc_init() - uzgadnianie sesji chronionej

pthread_join

Funkcja API gsk_secure_soc_close() - zamykanie sesji chronionej

Funkcja API gsk_environment_close() - zamykanie środowiska SSL

Funkcja API gsk_attribute_get_numeric_value() - uzyskiwanie informacji liczbowych o sesji chronionej lub środowisku SSL

Funkcja API gsk_secure_soc_write() - wysyłanie danych podczas sesji chronionej

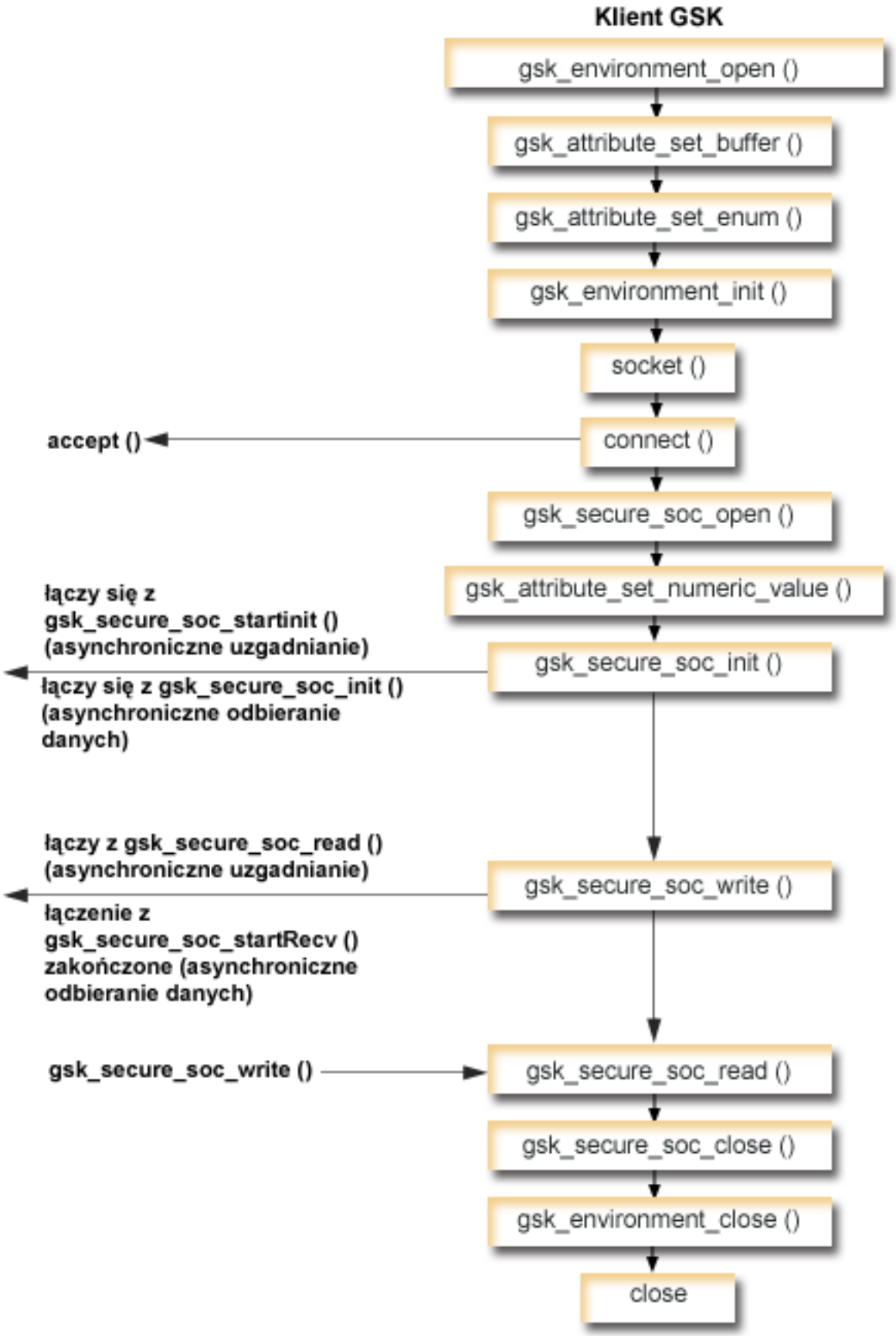
Funkcja API gsk_secure_soc_startInit() - uruchamianie operacji asynchronicznej w celu uzgadniania sesji chronionej

Funkcja API gsk_secure_soc_read() - odbieranie danych podczas sesji chronionej

Przykład: ustanawianie chronionego klienta za pomocą funkcji API Global Secure ToolKit

W tym przykładzie przedstawiono sposób ustanawiania klienta za pomocą funkcji API GSKit.

Na poniższym rysunku przedstawiono wywołania w kliencie chronionym używającym funkcji API GSKit.


Klient GSK
gsk_environment_open ()
gsk_attribute_set_buffer ()
gsk_attribute_set_enum ()
gsk_environment_init ()
socket ()
accept ()
connect ()
gsk_secure_soc_open ()
gsk_attribute_set_numeric_value ()
łączy się z gsk_secure_soc_startinit () (asynchroniczne uzgadnianie)
gsk_secure_soc_init ()
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_environment_close ()
close

Przebieg zdarzeń w gnieździe: klient GSKit

Ten przepływ przedstawia wywołania funkcji gniazd w poniższej aplikacji przykładowej. Tego klienta można używać z przykładowym serwerem GSKit i przykładowym chronionym serwerem GSKit z uzgadnianiem asynchronicznym.

1. Funkcja API gsk_environment_open() uzyskuje uchwyt środowiska SSL.
2. Co najmniej jedno wywołanie funkcji API gsk_attribute_set_xxxxx() w celu ustawienia atrybutów środowiska SSL. Minimum to wywołanie funkcji API 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. Należy ponadto ustawić typ aplikacji (klient lub serwer) i wartość GSK_SESSION_TYPE przy użyciu funkcji API gsk_attribute_set_enum().
3. Wywołanie funkcji API gsk_environment_init() w celu zainicjowania tego środowiska do przetwarzania SSL i określenia informacji o bezpieczeństwie dla wszystkich sesji SSL, które będą uruchamiane w tym środowisku.
4. Funkcja API socket() tworzy deskryptor gniazda. Następnie klient wywołuje funkcję API connect() w celu połączenia się z aplikacją serwera.
5. Funkcja API gsk_secure_soc_open() uzyskuje pamięć dla sesji chronionej, ustawia domyślne wartości atrybutów oraz zwraca uchwyt, który musi zostać zapisany i użyty podczas kolejnych wywołań funkcji API związanych z sesją chronioną.
6. Funkcja API gsk_attribute_set_numeric_value() przypisuje konkretne gniazdo do tej sesji chronionej.
7. Funkcja API gsk_secure_soc_init() uruchamia asynchroniczne uzgadnianie sesji chronionej przy użyciu zestawu atrybutów dla środowiska SSL i sesji chronionej.
8. Funkcja API 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 API gsk_secure_soc_startRecv(). W przykładzie serwera asynchronicznego zapisuje ona dane do zakończonej funkcji gsk_secure_soc_startInit().
9. Funkcja API gsk_secure_soc_read() odbiera komunikat od wątku procesu roboczego w ramach sesji chronionej.
10. Funkcja API gsk_secure_soc_close() kończy sesję chronioną.
11. Funkcja API gsk_environment_close() zamyka środowisko SSL.
12. Funkcja API close() kończy połączenie.

Uwaga: Korzystając z przykładów, użytkownik wyraża zgodę na warunki podane w sekcji “Licencja na kod oraz Informacje dotyczące kodu” na stronie 199.

```
/* Program klienta GSK używający ID aplikacji*/

/* Program zakłada, że ID aplikacji został           */
/* zarejestrowany, a certyfikat został przypisany    */
/* do ID aplikacji.                                  */
/*                                                   */
/* Brak parametrów, nieco 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];

  /*  Adres IP zapisany w kodzie (zmienić na   */
  /*  adres aplikacji serwera)                 */
  char addr[16] = "1.1.1.1";

  /*******************************************/
  /* Wszystkie komendy są uruchamiane w pętli  */
  /* do/while, dzięki czemu czyszczenie odbywa */
  /* się na końcu.                             */
  /*******************************************/
  do
  {
    /* otwórz 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 i kodem 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 i kodem 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 i kodem 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 zostanie wyłączony 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 i kodem błędu = %d.\n",
         rc,errno);
  printf("kod powrotu %d oznacza %s\n", rc, gsk_strerror(rc));
  break;
}

/* Określ, jakiego zestawu algorytmów szyfrowania użyć. Domyślnie jest */
/* włącz. domyślna lista szyfrowania. W tym przykł. zostanie ona użyta */
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 i kodem błędu = %d.\n",
         rc,errno);
  printf("kod powrotu %d oznacza %s\n", rc, gsk_strerror(rc));
  break;
}

/* Zainicjuj środowisko chronione       */
rc = errno = 0;
rc = gsk_environment_init(my_env_handle);
if (rc != GSK_OK)
{
  printf("Niepowodzenie gsk_environment_init() z kodem powrotu = %d i kodem 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 funkcji socket()");
  break;
}

/* połącz się 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 funkcji connect()");
  break;
}

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

/* powiąż gniazdo 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 uzgadniania 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 i kodem 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));

  /* wyślij komunikat do serwera w ramach sesji chronionej */
  strcpy(buff1,"Test of gsk_secure_soc_write \n\n");

  /* wyślij komunikat do klienta w ramach sesji chronionej */
  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;
    }
  }

  /* wyświetl wyniki 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));

  /* odbierz komunikat od klienta w ramach sesji chronionej */
  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;
  }

  /* wyświetl wyniki na ekranie */
  printf("Funkcja gsk_secure_soc_read() odebrała bajtów: %d, oto one:\n",
         amtRead);
```

```
    printf("%s\n",buff2);


  } while(FALSE);

  /* wyłącz obsługę SSL dla gniazda */
  if (my_session_handle != NULL)
    gsk_secure_soc_close(&my_session_handle);

  /* wyłącz środowisko SSL */
  if (my_env_handle != NULL)
    gsk_environment_close(&my_env_handle);

  /* zamknij połączenie */
  if (sd > -1)
    close(sd);

  return;
}
```

Pojęcia pokrewne

“Funkcje API z zestawu Global Secure ToolKit (GSKit)” na stronie 46
GSKit to zestaw programowalnych interfejsów, który umożliwia aplikacjom obsługę warstwy SSL.

Odsyłacze pokrewne

“Przykład: chroniony serwer GSKit z asynchronicznym odbieraniem danych” na stronie 119
W tym przykładzie przedstawiono sposób ustanawiania chronionego serwera za pomocą funkcji API GSKit.

“Przykład: chroniony serwer GSKit z uzgadnianiem asynchronicznym” na stronie 129
Funkcja API gsk_secure_soc_startInit() umożliwia tworzenie chronionych aplikacji serwera, które obsługują żądania w sposób asynchroniczny.

Informacje pokrewne

Funkcja API socket() - tworzenie gniazd

Funkcja API close() - zamykanie deskryptora gniazda lub pliku

Funkcja API connect() - nawiązywanie połączenia lub ustanawianie adresu docelowego

Funkcja API gsk_environment_open() - pobierania uchwytu dla środowiska SSL

Funkcja API gsk_attribute_set_buffer() - ustawianie informacji znakowych dla sesji chronionej lub środowiska SSL

Funkcja API gsk_attribute_set_enum() - ustawianie informacji wyliczeniowych dla sesji chronionej lub środowiska SSL

Funkcja API gsk_environment_init() - inicjowanie środowiska SSL

Funkcja API gsk_secure_soc_open() - pobieranie uchwytu dla sesji chronionej

Funkcja API gsk_attribute_set_numeric_value() - ustawianie informacji liczbowych dla sesji chronionej lub środowiska SSL

Funkcja API gsk_secure_soc_init() - uzgadnianie sesji chronionej

Funkcja API gsk_secure_soc_close() - zamykanie sesji chronionej

Funkcja API gsk_environment_close() - zamykanie środowiska SSL

Funkcja API gsk_secure_soc_write() - wysyłanie danych podczas sesji chronionej

Funkcja API gsk_secure_soc_startInit() - uruchamianie operacji asynchronicznej w celu uzgadniania sesji chronionej

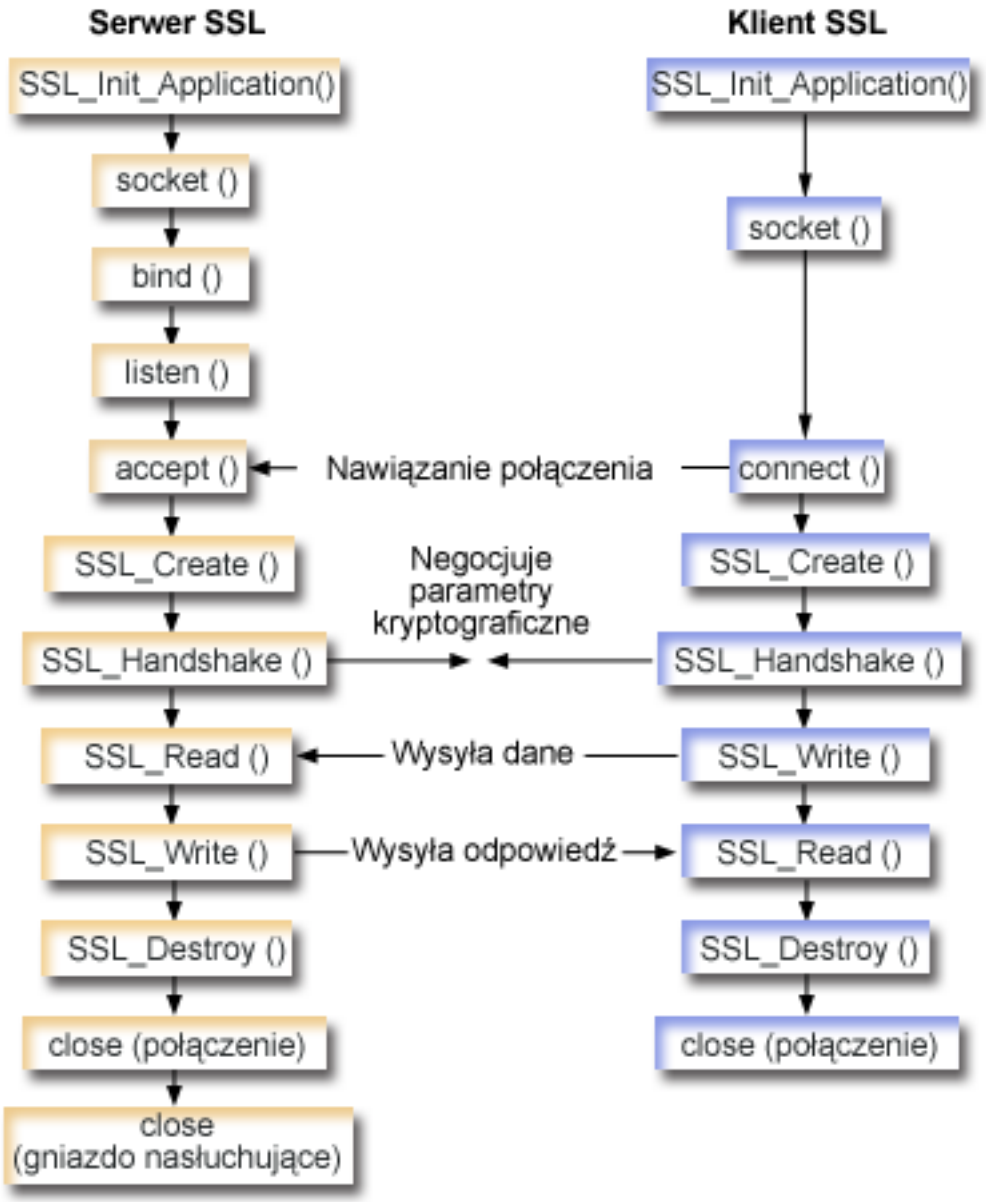
Funkcja API gsk_secure_soc_startRecv() - uruchamianie asynchronicznej operacji odbierania podczas sesji chronionej

Funkcja API gsk_secure_soc_read() - odbieranie danych podczas sesji chronionej

Przykład: ustanawianie chronionego serwera za pomocą funkcji API SSL_

Do tworzenia aplikacji chronionych można używać nie tylko funkcji API GSKit, lecz także funkcji API SSL_. Funkcje API SSL_ występują tylko w systemie operacyjnym i5/OS.

Na poniższym rysunku przedstawiono funkcje API gniazd i API SSL_ używane do tworzenia serwera chronionego.



Przebieg zdarzeń w gnieździe: chroniony serwer używający funkcji API SSL_

Poniżej opisano relacje między funkcjami API umożliwiającymi pracę serwera SSL i jego komunikację z klientem SSL:

1. Wywołanie funkcji API SSL_Init() lub SSL_Init_Application() w celu zainicjowania środowiska pracy dla przetwarzania SSL i w celu określenia informacji o bezpieczeństwie SSL dla wszystkich sesji SSL, które będą uruchamiane w bieżącym zadaniu. Należy użyć tylko jednej z wymienionych funkcji API. Zalecane jest użycie funkcji API SSL_Init_Application().

   Uwaga: Przedstawiony 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() w celu uaktywnienia połączenia dla programu serwerowego.
4. Serwer wywołuje funkcję SSL_Create() w celu włączenia obsługi SSL dla podłączonego gniazda.
5. Serwer wywołuje funkcję SSL_Handshake() w celu zainicjowania negocjacji parametrów szyfrowania podczas uzgadniania SSL.
6. Serwer wywołuje funkcje SSL_Write() i SSL_Read() w celu wysyłania i odbierania danych.
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 API SSL_Init() lub SSL_Init_Application() w celu zainicjowania środowiska pracy dla przetwarzania SSL i w celu określenia informacji o bezpieczeństwie SSL dla wszystkich sesji SSL, które będą uruchamiane w bieżącym zadaniu. Należy użyć tylko jednej z wymienionych funkcji API. Zalecane jest użycie funkcji API SSL_Init_Application().

   Uwaga: Przedstawiony 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 klienckiego.
4. Klient wywołuje funkcję SSL_Create() w celu włączenia obsługi SSL dla podłączonego gniazda.
5. Klient wywołuje funkcję SSL_Handshake() w celu zainicjowania negocjacji parametrów szyfrowania podczas uzgadniania SSL.
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, tak aby została użyta rodzina adresów AF_INET6. Korzystając z przykładów, użytkownik wyraża zgodę na warunki podane w sekcji “Licencja na kod oraz Informacje dotyczące kodu” na stronie 199.

```
/* Program serwera SSL używający funkcji SSL_Init_Application */

/* Przyjmuje się, że ID aplikacji jest już            */
/* zarejestrowany i powiązany z certyfikatem.         */
/*                                                    */
/* Brak parametrów, nieco 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 <qsossl.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 funkcji socket()");
  return;
}

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

/* powiązanie z adresem 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 funkcji bind()");
  close(lsd);
  return;
}

/* uaktywnienie gniazda dla przychodzących połączeń klienta */
listen(lsd, 5);
if (rc < 0)
```

```
{
  perror("Niepowodzenie funkcji listen()");
  close(lsd);
  return;
}

/* akceptacja przychodzącego połączenia klienta */
al = sizeof(address);
sd = accept(lsd, (struct sockaddr *) &address, &al);
if (sd < 0)
{
  perror("Niepowodzenie funkcji 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 uzgadniania 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 sesji chronionej */
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 sesji chronionej */
buf_len = strlen(buff);
rc = SSL_Write(sslh, buff, buf_len);
```

```
    if (rc != buf_len)
    {
      if (rc < 0)
      {
        printf("Niepowodzenie funkcji 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;
}
```

Pojęcia pokrewne

“Funkcje API SSL_” na stronie 49
Funkcje API SSL_ umożliwiają tworzenie aplikacji używających gniazd chronionych w systemie operacyjnym i5/OS.

Odsyłacze pokrewne

“Przykład: ustanawianie chronionego klienta za pomocą funkcji API SSL_” na stronie 151
Przykład ten zawiera aplikację klienta używającą funkcji API SSL_ do komunikowania się z aplikacją serwera używającą funkcji API SSL_.

Informacje pokrewne

Funkcja API SSL_Init() - inicjowanie bieżącego zadania dla środowiska SSL

Funkcja API SSL_Init_Application() - inicjowanie bieżącego zadania dla przetwarzania SSL na podstawie identyfikatora aplikacji

Funkcja API socket() - tworzenie gniazd

Funkcja API listen() - nasłuchiwanie przychodzących żądań połączenia

Funkcja API bind() - ustawianie lokalnego adresu gniazda

Funkcja API accept() - oczekiwanie na żądanie i nawiązywanie połączenia

Funkcja API close() - zamykanie deskryptora gniazda lub pliku

Funkcja API connect() - nawiązywanie połączenia lub ustanawianie adresu docelowego

Funkcja API SSL_Create() - włączanie obsługi środowiska SSL dla podanego deskryptora gniazda

Funkcja API SSL_Destroy() - kończenie obsługi środowiska SSL dla podanej sesji SSL

Funkcja API SSL_Handshake() - inicjowanie protokołu uzgadniania środowiska SSL

Funkcja API SSL_Read() - odbieranie danych z deskryptora gniazda z włączoną obsługą SSL

Funkcja API SSL_Write() - zapisywanie danych do deskryptora gniazda z włączoną obsługą SSL

Przykład: ustanawianie chronionego klienta za pomocą funkcji API SSL_

Przykład ten zawiera aplikację klienta używającą funkcji API SSL_ do komunikowania się z aplikacją serwera używającą funkcji API SSL_.

Uwaga: Korzystając z przykładów, użytkownik wyraża zgodę na warunki podane w sekcji “Licencja na kod oraz Informacje dotyczące kodu” na stronie 199.

```
/* Program klienta SSL używający funkcji SSL_Init_Application */

/* Przyjmuje się, że ID aplikacji jest już            */
/* zarejestrowany i powiązany z certyfikatem.         */
/*                                                    */
/* Brak parametrów, nieco 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 <qsossl.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[16] = "1.1.1.1";

    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 funkcji 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 funkcji connect()");
  close(sd);
  return;
}

/* przygotowanie do wywołania uzgodnienia, ustawianie algorytmu */
sslh -> protocol = 0;
sslh -> timeout = 0;
sslh -> cipherSuiteList = &cipher;
sslh -> cipherSuiteListLen = 1;

/* inicjowanie uzgadniania 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 sesji chronionej */
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 funkcji 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 sesji chronionej */
    rc = SSL_Read(sslh, buff2, buf_len);
    if (rc < 0)
    {
      printf("Niepowodzenie funkcji 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;
}
```

Pojęcia pokrewne

“Funkcje API SSL_” na stronie 49
Funkcje API SSL_ umożliwiają tworzenie aplikacji używających gniazd chronionych w systemie operacyjnym i5/OS.

Odsyłacze pokrewne

“Przykład: ustanawianie chronionego serwera za pomocą funkcji API SSL_” na stronie 145
Do tworzenia aplikacji chronionych można używać nie tylko funkcji API GSKit, lecz także funkcji API SSL_. Funkcje API SSL_ występują tylko w systemie operacyjnym i5/OS.

Przykład: używanie funkcji gethostbyaddr_r() dla wątkowo bezpiecznych procedur sieciowych

Przedstawiony program przykładowy używa funkcji API gethostbyaddr_r(). Wszystkie pozostałe procedury, których nazwy kończą się przyrostkiem *_r*, mają podobną semantykę i są również wątkowo bezpieczne.

Ten program przykładowy pobiera adres IP w postaci dziesiętnej z kropkami i drukuje nazwę hosta.

Uwaga: Korzystając z przykładów, użytkownik wyraża zgodę na warunki podane w sekcji “Licencja na kod oraz Informacje dotyczące kodu” na stronie 199.

```
/*********************************************************/
/* 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 postaci     */
/* dziesiętnej z kropkami. Jeśli nazwa hosta będzie      */
/* znaleziona, zostanie wyświetlona; w przeciwnym        */
/* razie wyświetli się 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 przekazanych 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 się nie powiodło)                      */
    /*   0 (wywołanie się powiodło)                          */
    /*********************************************************/
    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 przez       */
    /* 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);
 }
```

Pojęcia pokrewne

“Ochrona wątków” na stronie 57
Funkcja jest wątkowo bezpieczna (zapewnia ochronę wątków), jeśli w ramach tego samego procesu można ją uruchomić jednocześnie w wielu wątkach. Funkcję uznaje się za wątkowo bezpieczną tylko wtedy, gdy wszystkie funkcje przez nią wywoływane są również wątkowo bezpieczne. Funkcje API gniazd obejmują wątkowo bezpieczne funkcje systemowe i funkcje sieciowe.

Odsyłacze pokrewne

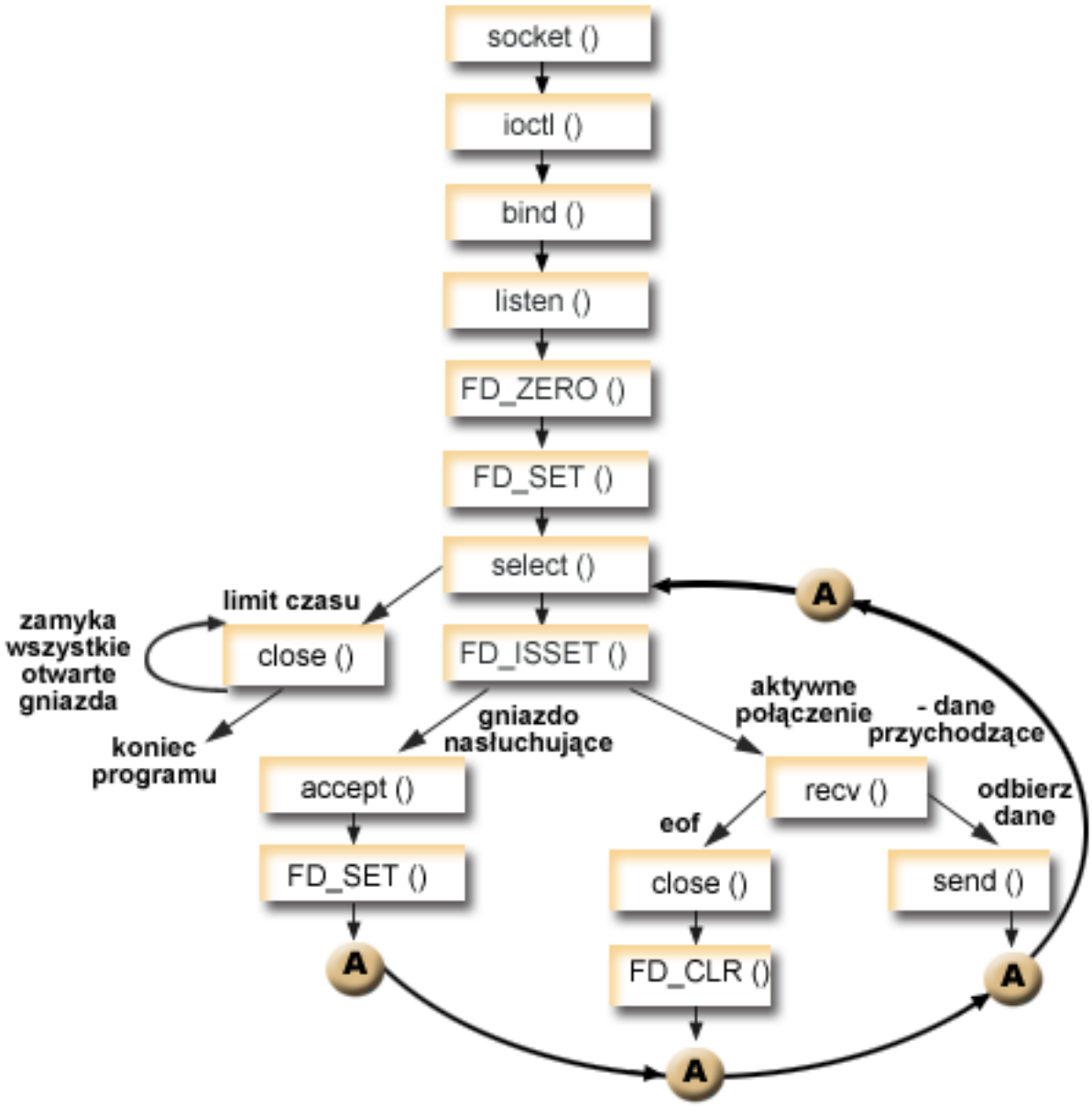
“Funkcje sieciowe gniazd” na stronie 63
Funkcje sieciowe gniazd umożliwiają aplikacjom uzyskiwanie informacji od hostów, protokołów, usług oraz plików sieciowych.

Informacje pokrewne

Funkcja API gethostbyaddr_r() - pobieranie informacji o hoście dla podanego adresu IP

Przykład: nieblokujące operacje we/wy i funkcja select()

W tym przykładzie przedstawiono aplikację serwera, która wykorzystuje nieblokujące operacje we/wy i funkcję API select().



Przebieg zdarzeń w gnieździe: aplikacja serwera używająca nieblokujących operacji we/wy i funkcji API select

W przykładzie użyto następujących wywołań funkcji:

1. Funkcja API socket() zwraca deskryptor gniazda reprezentujący punkt końcowy. Instrukcja ta informuje również, że dla tego gniazda zostanie użyta rodzina adresów INET (Internet Protocol) i protokół transportowy TCP (SOCK_STREAM).
2. Funkcja API ioctl() umożliwia aplikacji ponowne użycie adresu lokalnego w razie zrestartowania serwera przed upływem wymaganego czasu oczekiwania. W tym przykładzie ustawia gniazdo w trybie nieblokującym. Wszystkie gniazda dla połączeń przychodzących również są nieblokujące, ponieważ będą dziedziczyć ten stan od gniazda nasłuchującego.
3. Po utworzeniu deskryptora gniazda funkcja API bind() pobiera unikalną nazwę gniazda.
4. Funkcja API listen() pozwala serwerowi na przyjmowanie połączeń przychodzących od klientów.

5. Serwer akceptuje żądanie połączenia przychodzącego za pomocą funkcji API accept(). Wywołanie funkcji API accept() zostanie zablokowane na nieokreślony czas oczekiwania na połączenie przychodzące.
6. Funkcja API select() powoduje, że proces oczekuje na wystąpienie zdarzenia, po którym działanie zostaje aktywowane. W tym przykładzie funkcja API select () zwraca liczbę odpowiadającą deskryptorom gniazd, gotowym do przetwarzania.

   0 Wskazuje, że następuje przekroczenie limitu czasu procesu. W tym przykładzie limit czasu ustawiono na 30 sekund.

   -1 Wskazuje, że proces zakończył się niepowodzeniem.

   1 Wskazuje, że tylko jeden deskryptor jest gotowy do przetworzenia. W tym przykładzie zwrócenie wartości 1 powoduje, że FD_ISSET i kolejne wywołania gniazd zostaną zakończone tylko raz.

   n Wskazuje, że na przetworzenie oczekuje 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 API accept() i recv() kończą działanie po zwróceniu wartości EWOULDBLOCK.
8. Funkcja API send() odsyła dane do klienta.
9. Funkcja API close() zamyka wszelkie otwarte deskryptory gniazd.

Uwaga: Korzystając z przykładów, użytkownik wyraża zgodę na warunki podane w sekcji “Licencja na kod oraz Informacje dotyczące kodu” na stronie 199.

```
#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;

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

   /*************************************************************/
   /* Pozwól na ponowne użycie deskryptora gniazda              */
   /*************************************************************/
   rc = setsockopt(listen_sd, SOL_SOCKET,  SO_REUSEADDR,
                   (char *)&on, sizeof(on));
```

```
if (rc < 0)
{
   perror("Niepowodzenie funkcji setsockopt()");
   close(listen_sd);
   exit(-1);
}

/*************************************************************/
/* Ustaw gniazdo jako nieblokujące. Wszystkie gniazda        */
/* 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 funkcji 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 funkcji bind()");
   close(listen_sd);
   exit(-1);
}

/*************************************************************/
/* Ustaw parametr backlog funkcji listen.                    */
/*************************************************************/
rc = listen(listen_sd, 32);
if (rc < 0)
{
   perror("Niepowodzenie funkcji listen()");
   close(listen_sd);
   exit(-1);
}

/*************************************************************/
/* Inicjuje główny fd_set                                    */
/*************************************************************/
FD_ZERO(&master_set);
max_sd = listen_sd;
FD_SET(listen_sd, &master_set);

/*************************************************************/
/* Zainicjuj strukturę limitu czasu z wartością 3 minuty.    */
/* Brak aktywności w tym czasie spowoduje zakończ. programu. */
/*************************************************************/
timeout.tv_sec  = 3 * 60;
timeout.tv_usec = 0;

/*************************************************************/
/* Pętla oczekiwania na połączenia lub dane                  */
/* przychodzące dla dowolnego połączonego gniazda.           */
/*************************************************************/
do
{
```

```
  /*********************************************************/
  /* Skopiuj główną wartość fd_set do wart. roboczej fd_set */
  /*********************************************************/
  memcpy(&working_set, &master_set, sizeof(master_set));

  /*********************************************************/
  /* Wywołuje select() i czeka 5 min na zakończenie.       */
   /*********************************************************/
  printf("  Oczekiwanie na wywołanie funkcji select()\n");
  rc = select(max_sd + 1, &working_set, NULL, NULL, &timeout);

  /*********************************************************/
  /* Sprawdź, czy wywołanie funkcji select się powiodło.   */
  /*********************************************************/
  if (rc < 0)
  {
     perror("Niepowodzenie funkcji select()");
     break;
  }

  /*********************************************************/
  /* Sprawdź, czy upłynął 5-minutowy czas oczekiwania.     */
  /*********************************************************/
  if (rc == 0)
  {
     printf("Przekroczenie limitu czasu funkcji select(). Zakończ program.\n");
     break;
  }

  /*********************************************************/
  /* Co najmniej jeden deskryptor jest czytelny.           */
  /* Sprawdź, 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 ta jest powtarzana, więc*/
        /* po znalezieniu wszystkich gotowych deskryptorów  */
        /* można zakończyć przeszukiwanie zestawu           */
        /* roboczego.                                       */
        /**************************************************/
        desc_ready -= 1;

        /**************************************************/
        /* Sprawdź, czy gniazdo jest nasłuchujące.          */
        /**************************************************/
        if (i == listen_sd)
        {
           printf("Gniazdo nasłuchujące jest czytelne\n");
           /***********************************************/
           /* Akceptuj wszystkie połączenia przychodzące,   */
           /* które znajdują się w kolejce gniazda          */
           /* nasłuchującego, przed powrotem do pętli       */
           /* i ponownym wywołaniem funkcji select.         */
           /***********************************************/
           do
           {
              /********************************************/
              /* Akceptuj każde połączenie przychodzące.    */
              /* Jeśli akceptowanie nie powiedzie się       */
```

```
      /* z wartością EWOULDBLOCK, to zostały        */
      /* zaakceptowane wszystkie. Każde inne        */
      /* niepowodzenie akceptowania zakończy        */
      /* działanie serwera.                                    */
      /*******************************************/
      new_sd = accept(listen_sd, NULL, NULL);
      if (new_sd < 0)
      {
         if (errno != EWOULDBLOCK)
         {
            perror("Niepowodzenie funkcji accept()");
            end_server = TRUE;
         }
         break;
      }

      /*******************************************/
      /* Dodaj nowe połączenie przychodzące                */
      /* 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;

      /*******************************************/
      /* Powróć do pętli i zaakceptuj nowe          */
      /* połączenie przychodzące.                   */
      /*******************************************/
   } while (new_sd != -1);
}

/*****************************************************/
/* Nie jest to gniazdo nasłuchujące, dlatego        */
/* istniejące połączenie musi być czytelne          */
/*****************************************************/
else
{
   printf("Deskryptor %d jest czytelny\n", i);
   close_conn = FALSE;
   /*******************************************/
   /* Odbierz wszystkie dane przychodzące do        */
   /* tego gniazda przed powrotem w pętli i ponownym*/
   /* wywołaniem funkcji select.                 */
   /*******************************************/
   do
   {
      /*******************************************/
      /* Odbieraj dane dla tego połączenia aż do    */
      /* wystąpienia niepowodzenia funkcji recv     */
      /* 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 funkcji recv()");
            close_conn = TRUE;
         }
         break;
      }

      /*******************************************/
      /* Sprawdź, 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);

                  /*********************************************/
                  /* Odeślij dane do klienta                   */
                  /*********************************************/
                  rc = send(i, buffer, len, 0);
                  if (rc < 0)
                  {
                     perror("Niepowodzenie funkcji 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 są nadal 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;
                  }
               }
            } /* Zakończenie istniejącego połączenia jest czytelne   */
         } /* Koniec "if (FD_ISSET(i, &working_set))" */
      } /* Koniec pętli poprzez wybierane deskryptory  */

   } while (end_server == FALSE);

   /*************************************************************/
   /* Wyczyść wszystkie otwarte gniazda                         */
   /*************************************************************/
   for (i=0; i <= max_sd; ++i)
   {
      if (FD_ISSET(i, &master_set))
         close(i);
   }
}
```

Pojęcia pokrewne

“Nieblokujące operacje we/wy” na stronie 57
Kiedy aplikacja wywoła jedną z wejściowych funkcji API gniazd, a nie będzie żadnych danych do odczytu, to funkcja API się zablokuje i nie zakończy działania aż do chwili pojawienia się danych do odczytu.

“Multipleksowanie we/wy - funkcja select()” na stronie 63
Zalecane jest używanie asynchroniczne operacji we/wy, ponieważ zapewniają bardziej efektywną metodę maksymalizowania zasobów aplikacji niż funkcja API select(). Konkretny projekt aplikacji może jednak dopuszczać użycie funkcji API select().

Odsyłacze pokrewne

“Zalecenia dotyczące projektowania aplikacji używających gniazd” na stronie 84
Przed przystąpieniem do pracy z aplikacją używającą gniazd należy ocenić jej wymagania funkcjonalne, cele i potrzeby. Należy również rozważyć wymagania aplikacji dotyczące wydajności oraz jej wpływ na zasoby systemu.

“Przykład: ogólny program klienta” na stronie 109
W przykładzie użyto kodu typowych zadań klienta. Zadanie klienta używa funkcji socket(), connect(), send(), recv() i close().

Informacje pokrewne

Funkcja API accept() - oczekiwanie na żądanie i nawiązywanie połączenia

Funkcja API recv() - odbieranie danych

Funkcja API ioctl() - wykonywanie żądań sterowania operacjami we/wy

Funkcja API send() - wysyłanie danych

Funkcja API listen() - nasłuchiwanie przychodzących żądań połączenia

Funkcja API close() - zamykanie deskryptora gniazda lub pliku

Funkcja API socket() - tworzenie gniazd

Funkcja API bind() - ustawianie lokalnego adresu gniazda

Funkcja API select() - oczekiwanie na wydarzenia w wielu gniazdach

Używanie funkcji poll() zamiast funkcji select()

Funkcja API poll() wchodzi w skład ujednoliconej specyfikacji standardu Unix oraz standardu UNIX 95/98. Funkcja poll() realizuje te same zadania co funkcja select(). Jedyna różnica między nimi dotyczy interfejsu udostępnianego programowi wywołującemu.

Funkcja select() wymaga, aby aplikacja przesyłała dane jako szereg bitów, przy czym każdy bit oznacza numer deskryptora. Jeśli numery deskryptorów są bardzo duże, może to spowodować przepełnienie przydzielonych 30 KB pamięci i konieczność wielokrotnego powtórzenia tego procesu. Może to obniżyć wydajność.

Funkcja poll() umożliwia aplikacji wysłanie szeregu struktur zamiast szeregu bitów. Ponieważ każda struktura pollfd może zawierać do 8 bajtów, aplikacja musi wysłać tylko jedną strukturę dla każdego deskryptora, nawet jeśli numery deskryptora są bardzo duże.

Przebieg zdarzeń w gnieździe: serwer używający funkcji poll()

W przykładzie użyto następujących wywołań funkcji:

1. Funkcja API socket() zwraca deskryptor gniazda reprezentujący punkt końcowy. Instrukcja informuje również, że dla tego gniazda jest używana rodzina adresów AF_INET i protokół transportowy TCP (SOCK_STREAM).
2. Funkcja API setsockopt() umożliwia aplikacji ponowne użycie adresu lokalnego w razie zrestartowania serwera przed upływem wymaganego czasu oczekiwania.
3. Funkcja API ioctl() ustawia gniazdo w trybie nieblokującym. Wszystkie gniazda dla połączeń przychodzących również są nieblokujące, ponieważ będą dziedziczyć ten stan od gniazda nasłuchującego.
4. Po utworzeniu deskryptora gniazda funkcja API bind() pobiera unikalną nazwę gniazda.
5. Wywołanie funkcji API listen() pozwala serwerowi na przyjmowanie połączeń przychodzących z klientów.
6. Funkcja API select() powoduje, że proces oczekuje na wystąpienie zdarzenia, po którym kontynuuje działanie. Funkcja API poll() może zwracać następujące wartości.

   0 Wskazuje, że następuje przekroczenie limitu czasu procesu. W tym przykładzie limit czasu ustawiono na 3 minuty (w milisekundach).

-1 Wskazuje, że proces zakończył się niepowodzeniem.

1 Wskazuje, że tylko jeden deskryptor gniazda jest gotowy do przetworzenia; zostanie on przetworzony tylko wtedy, gdy będzie to gniazdo nasłuchujące.

1++ Wskazuje, że na przetworzenie oczekuje wiele deskryptorów. Funkcja poll() umożliwia symultaniczne połączenie ze wszystkimi deskryptorami w kolejce gniazda nasłuchującego.

7. Funkcje API accept() i recv() kończą działanie po zwróceniu wartości EWOULDBLOCK.
8. Funkcja API send() odsyła dane do klienta.
9. Funkcja API close() zamyka wszelkie otwarte deskryptory gniazd.

Uwaga: Korzystając z przykładów, użytkownik wyraża zgodę na warunki podane w sekcji “Licencja na kod oraz Informacje dotyczące kodu” na stronie 199.

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/ioctl.h>
#include <sys/poll.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    len, rc, on = 1;
  int    listen_sd = -1, new_sd = -1;
  int    desc_ready, end_server = FALSE, compress_array = FALSE;
  int    close_conn;
  char   buffer[80];
  struct sockaddr_in   addr;
  int    timeout;
  struct pollfd fds[200];
  int    nfds = 1, current_size = 0, i, j;

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

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

  /*************************************************************/
  /* Ustaw gniazdo jako nieblokujące. Wszystkie gniazda        */
  /* 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 funkcji 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 funkcji bind()");
  close(listen_sd);
  exit(-1);
}

/*************************************************************/
/* Ustaw parametr backlog funkcji listen.                    */
/*************************************************************/
rc = listen(listen_sd, 32);
if (rc < 0)
{
  perror("Niepowodzenie funkcji listen()");
  close(listen_sd);
  exit(-1);
}

/*************************************************************/
/* Zainicjuj strukturę pollfd                                */
/*************************************************************/
memset(fds, 0 , sizeof(fds));

/*************************************************************/
/* Ustaw początkowe gniazdo nasłuchujące                     */
/*************************************************************/
fds[0].fd = listen_sd;
fds[0].events = POLLIN;
/*************************************************************/
/* Ustaw limit czasu na 3 minuty. Brak                       */
/* aktywności w tym czasie spowoduje zakończenie programu.   */
/* Limit czasu jest liczony w milisekundach.                  */
/*************************************************************/
timeout = (3 * 60 * 1000);

/*************************************************************/
/* Pętla oczekiwania na połączenia lub dane                  */
/* przychodzące dla dowolnego połączonego gniazda.           */
/*************************************************************/
do
{
  /***********************************************************/
  /* Wywołaj select() i zaczekaj 3 min na zakończenie.       */
  /***********************************************************/
  printf("Oczekiwanie na wywołanie funkcji select()...\n");
  rc = poll(fds, nfds, timeout);

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

```
/* Sprawdź, czy wywołanie funkcji poll się                */
/* powiodło.                                              */
/**********************************************************/
if (rc < 0)
{
  perror("Niepowodzenie funkcji poll()");
  break;
}

/**********************************************************/
/* Sprawdź, czy upłynęły 3 min czasu oczekiwania.         */
/**********************************************************/
if (rc == 0)
{
  printf("Przekroczenie limitu czasu funkcji poll(). Zakończ program.\n");
  break;
}


/**********************************************************/
/* Co najmniej jeden deskryptor jest czytelny.            */
/* Sprawdź, który.                                        */
/**********************************************************/
current_size = nfds;
for (i = 0; i < current_size; i++)
{
  /********************************************************/
  /* Znajdź deskryptory, które zwróciły                   */
  /* wartość POLLIN, i określ, czy jest to połączenie     */
  /* nasłuchujące, czy aktywne.                           */
  /********************************************************/
  if(fds[i].revents == 0)
    continue;

  /********************************************************/
  /* Jeśli pole revents nie ma wartości POLLIN, jest to   */
  /* rezultat nieoczekiwany, należy więc sporządzić       */
  /* protokół i zak. działanie serwera.                          */
  /********************************************************/
  if(fds[i].revents != POLLIN)
  {
    printf("  Błąd! revents = %d\n", fds[i].revents);
    end_server = TRUE;
    break;

  }
  if (fds[i].fd == listen_sd)
  {
    /******************************************************/
    /* Deskryptor nasłuchujący jest czytelny.                */
    /******************************************************/
    printf("Gniazdo nasłuchujące jest czytelne\n");

    /******************************************************/
    /* Akceptuj wszystkie połączenia przychodzące,        */
    /* które znajdują się w kolejce gniazda               */
    /* nasłuchującego, przed ponownym wywołaniem          */
    /* funkcji poll.                                      */
    /******************************************************/
    do
    {
      /****************************************************/
      /* Akceptuj każde połączenie przychodzące.          */
      /* Jeśli akceptowanie nie powiedzie się             */
      /* z wartością EWOULDBLOCK, to zostały              */
      /* zaakceptowane wszystkie. Każde inne              */
      /* niepowodzenie akceptowania zakończy              */
```

```
    /* działanie serwera.                                         */
    /************************************************************/
    new_sd = accept(listen_sd, NULL, NULL);
    if (new_sd < 0)
    {
      if (errno != EWOULDBLOCK)
      {
        perror("Niepowodzenie funkcji accept()");
        end_server = TRUE;
      }
      break;
    }

    /************************************************************/
    /* Dodaj nowe połączenie przychodzące                       */
    /* do struktury pollfd                                      */
    /************************************************************/
    printf("Nowe połączenie przychodzące - %d\n", new_sd);
    fds[nfds].fd = new_sd;
    fds[nfds].events = POLLIN;
    nfds++;

    /************************************************************/
    /* Powróć do pętli i zaakceptuj nowe                        */
    /* połączenie przychodzące.                                 */
    /************************************************************/
  } while (new_sd != -1);
}

/************************************************************/
/* Nie jest to gniazdo nasłuchujące, dlatego                */
/* istniejące połączenie musi być czytelne                  */
/************************************************************/

else
{
  printf("Deskryptor %d jest czytelny\n", i);
  close_conn = FALSE;
  /************************************************************/
  /* Odbierz wszystkie dane przychodzące do                   */
  /* tego gniazda przed powrotem w pętli i ponownym           */
  /* wywołaniem funkcji poll.                                 */
  /************************************************************/

  do
  {
    /************************************************************/
    /* Odbieraj dane dla tego połączenia aż do                  */
    /* wystąpienia niepowodzenia funkcji recv                   */
    /* z wartością EWOULDBLOCK. Jeśli wystąpi inne              */
    /* niepowodzenie, połączenie zostanie zamknięte.            */
    /************************************************************/
    rc = recv(fds[i].fd, buffer, sizeof(buffer), 0);
    if (rc < 0)
    {
      if (errno != EWOULDBLOCK)
      {
        perror("Niepowodzenie funkcji recv()");
        close_conn = TRUE;
      }
      break;
    }

    /************************************************************/
    /* Sprawdź, 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);

      /*****************************************************/
      /* Odeślij dane do klienta                           */
      /*****************************************************/
      rc = send(fds[i].fd, buffer, len, 0);
      if (rc < 0)
      {
        perror("Niepowodzenie funkcji send()");
        close_conn = TRUE;
        break;
      }

    } while (TRUE);

    /*******************************************************/
    /* Jeśli opcja close_conn została włączona,            */
    /* trzeba wyczyścić aktywne połączenie. Procedura      */
    /* czyszcząca obejmuje usunięcie deskryptora.          */
    /*                                                     */
    /*******************************************************/
    if (close_conn)
    {
      close(fds[i].fd);
      fds[i].fd = -1;
      compress_array = TRUE;
    }


  } /* Zakończenie istniejącego połączenia jest czytelne   */
}   /* Zakończenie pętli poprzez deskryptory do odpytania  */

/************************************************************/
/* Jeśli opcja compress array została włączona, należy      */
/* skompresować tablice i zmniejszyć liczbę deskryptorów    */
/* plików. Nie jest koniecznie ponowne przenoszenie pól     */
/* events i revents, gdyż pole events zawsze będzie miało   */
/* wartość POLLIN, a revents będzie polem danych            */
/* wyjściowych.                                             */
/************************************************************/
if (compress_array)
{
  compress_array = FALSE;
  for (i = 0; i < nfds; i++)
  {
    if (fds[i].fd == -1)
    {
      for(j = i; j < nfds; j++)
      {
        fds[j].fd = fds[j+1].fd;
      }
      nfds--;
    }
  }
}
```

```
  } while (end_server == FALSE); /* Koniec działania serwera.    */

  /*************************************************************/
  /* Wyczyść wszystkie otwarte gniazda                         */
  /*************************************************************/
  for (i = 0; i < nfds; i++)
  {
    if(fds[i].fd >= 0)
      close(fds[i].fd);
  }
}
```

Informacje pokrewne

Funkcja API accept() - oczekiwanie na żądanie i nawiązywanie połączenia

Funkcja API recv() - odbieranie danych

Funkcja API ioctl() - wykonywanie żądań sterowania operacjami we/wy

Funkcja API send() - wysyłanie danych

Funkcja API listen() - nasłuchiwanie przychodzących żądań połączenia

Funkcja API close() - zamykanie deskryptora gniazda lub pliku

Funkcja API socket() - tworzenie gniazd

Funkcja API bind() - ustawianie lokalnego adresu gniazda

Funkcja API setsockopt_ioctl() - ustawianie opcji gniazd

Funkcja API poll() - oczekiwania na wydarzenia w wielu deskryptorach

Przykład: używanie sygnałów z blokującymi funkcjami API gniazd

Sygnały mogą przekazywać powiadomienie o tym, że doszło do zablokowania procesu lub aplikacji. Udostępniają także limit czasu blokowania procesów.

W tym przykładzie sygnał pojawia się po pięciu sekundach od wywołania funkcji API accept(). Wywołanie to zwykle blokuje proces na czas nieokreślony, lecz ponieważ został ustawiony alarm, wywołanie zostanie zablokowane jedynie na pięć sekund. Ponieważ zablokowane programy mogą zmniejszać wydajność aplikacji lub serwera, użycie sygnałów może ograniczyć te negatywne skutki. W tym przykładzie przedstawiono sposób używania sygnałów z blokującymi funkcjami API gniazd.

Uwaga: Zamiast modelu konwencjonalnego preferowane jest użycie asynchronicznych operacji we/wy w modelu serwera z obsługą wątków.

Przebieg zdarzeń w gnieździe: używanie sygnałów z blokowaniem gniazda

Poniższa sekwencja wywołań funkcji API przedstawia sposób użycia sygnałów do powiadamiania aplikacji o tym, że gniazdo jest nieaktywne.

1. Funkcja API socket() zwraca deskryptor gniazda reprezentujący punkt końcowy. Instrukcja informuje również, że dla tego gniazda jest używana rodzina adresów AF_INET i protokół transportowy TCP (SOCK_STREAM).
2. Po utworzeniu deskryptora gniazda funkcja API bind() pobiera unikalną nazwę gniazda. W tym przykładzie numer portu nie jest określony, ponieważ aplikacja klienta nie nawiązuje połączenia z gniazdem. Tego fragmentu kodu można użyć w innych programach serwera, które używają blokowania takich funkcji API jak accept().
3. Funkcja API listen() wskazuje na gotowość do zaakceptowania żądań połączenia wysyłanych przez klienta. Po wywołaniu funkcji API listen() jest ustawiany alarm, który uruchomi się po pięciu sekundach. Ten alarm lub sygnał informuje użytkownika o zablokowaniu wywołania funkcji accept().
4. Funkcja API accept() akceptuje żądanie połączenia od klienta. Wywołanie to zwykle blokuje proces na czas nieokreślony, lecz ponieważ został ustawiony alarm, wywołanie zostanie zablokowane jedynie na pięć sekund. Kiedy alarm się włączy, funkcja accept zakończy się z kodem powrotu -1 i kodem błędu EINTR.
5. Funkcja API close() zamyka wszelkie otwarte deskryptory gniazd.

Uwaga: Korzystając z przykładów, użytkownik wyraża zgodę na warunki podane w sekcji “Licencja na kod oraz Informacje dotyczące kodu” na stronie 199.

```
/*********************************************************************/
/* 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;

/*********************************************************************/
/* Utwórz gniazdo strumieniowe 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);
   }

/*********************************************************************/
/* Powiąż gniazdo. Numer portu nie został podany, 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("Niepowodzenie funkcji bind()");
      close(sd);
      return(-2);
   }

/*********************************************************************/
/* Wykonanie czynności nasłuchiwania przez gniazdo.               */
/*********************************************************************/
   printf("Ustawienie parametru backlog nasłuchiwania\n");
   rc = listen(sd, 5);
   if (rc != 0)
   {
      perror("Niepowodzenie funkcji listen()");
      close(sd);
      return(-3);
   }

/*********************************************************************/
/* Ustawienie alarmu, który włączy się po pięciu sekundach.       */
/*********************************************************************/
   printf("\nUstawienie alarmu, który włączy się po 5 s. Alarm spowoduje,\n");
   printf("że zablokowana funkcja accept() zwróci wartość -1 i kod błędu 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));

/*****************************************************************/
/* Wywołaj funkcję accept. Normalnie wywołanie to blokowałoby    */
/* przez czas nieograniczony, ale alarm powoduje ograniczenie    */
/* blokowania do 5 sekund. Gdy alarm się włączy, funkcja         */
/* accept zakończy się z wartością -1 i kodem błędu 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);
}
```

Pojęcia pokrewne

“Sygnały” na stronie 59
Aplikacja może zażądać asynchronicznego powiadomienia (żądanie wysłania przez system *sygnału*) o wystąpieniu warunku, na który czeka.

“Asynchroniczne operacje we/wy” na stronie 42
Funkcje API asynchronicznych operacji we/wy udostępniają metodę realizacji wątkowych modeli klient/serwer w celu zapewnienia wysoko współbieżnych operacji we/wy z efektywnym wykorzystaniem pamięci.

Odsyłacze pokrewne

“Zgodność z Berkeley Software Distribution (BSD)” na stronie 66
Gniazda są interfejsem systemu BSD.

“Przykład: korzystanie z asynchronicznych operacji we/wy” na stronie 112
Aplikacja tworzy port zakończenia operacji we/wy za pomocą funkcji API QsoCreateIOCompletionPort(). Funkcja ta zwraca uchwyt, za pomocą którego można zaplanować asynchroniczne żądania we/wy i oczekiwać na ich zakończenie.

Informacje pokrewne

Funkcja API accept() - oczekiwanie na żądanie i nawiązywanie połączenia

Funkcja API listen() - nasłuchiwanie przychodzących żądań połączenia

Funkcja API close() - zamykanie deskryptora gniazda lub pliku

Funkcja API socket() - tworzenie gniazd

Funkcja API bind() - ustawianie lokalnego adresu gniazda

Przykłady: rozsyłanie grupowe za pomocą rodziny adresów AF_INET

Rozsyłanie grupowe IP pozwala aplikacjom na wysłanie pojedynczego datagramu IP, który zostanie odebrany przez grupę hostów w sieci.

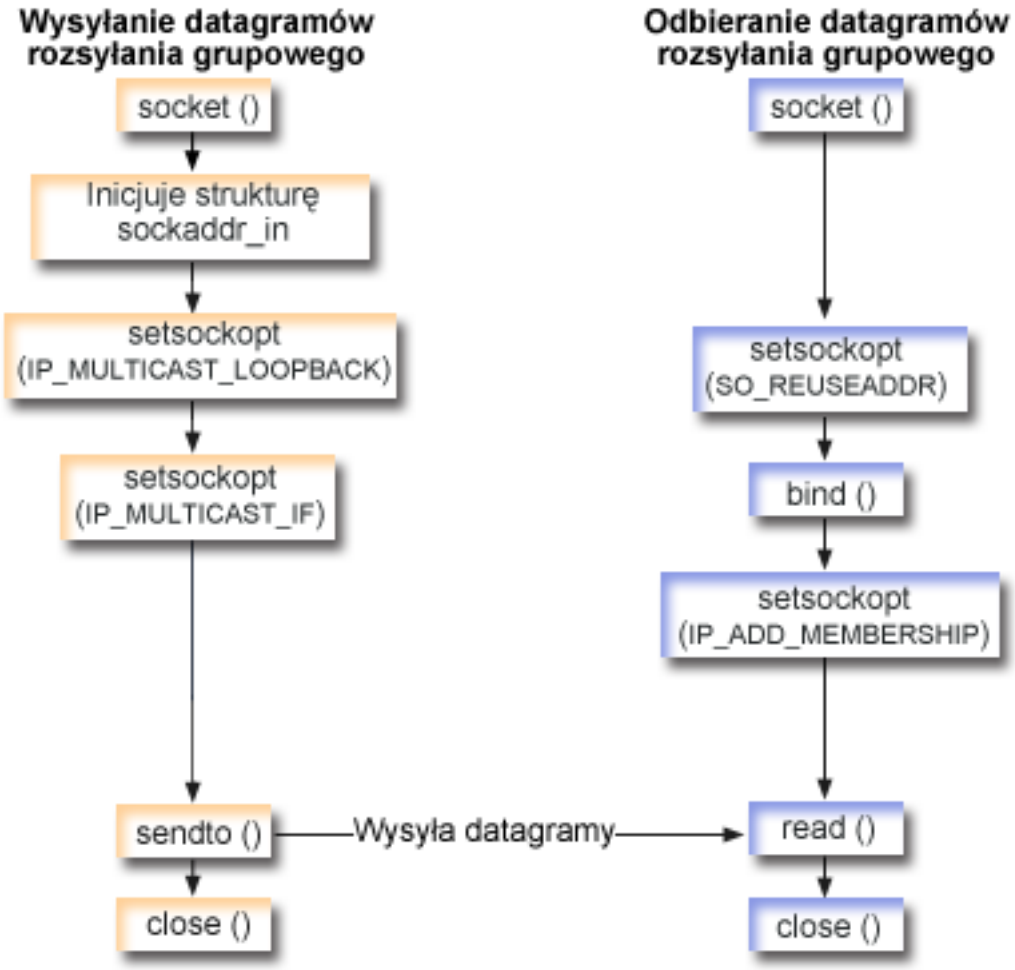
Uwaga: Korzystając z przykładów, użytkownik wyraża zgodę na warunki podane w sekcji “Licencja na kod oraz Informacje dotyczące kodu” na stronie 199.

Hosty należące do grupy mogą znajdować się w tej samej podsieci lub w różnych podsieciach połączonych routerami obsługującymi rozsyłanie grupowe. Hosty mogą dołączać do grup i opuszczać je w każdej chwili. Nie ma ograniczeń dotyczących położenia ani liczby elementów grupy hostów. Grupę hostów identyfikuje klasa D adresów IP w zakresie od 224.0.0.1 do 239.255.255.255.

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

- IP_ADD_MEMBERSHIP: dołącza do podanej grupy rozsyłania.
- IP_DROP_MEMBERSHIP: opuszcza podaną grupę rozsyłania.
- IP_MULTICAST_IF: konfiguruje interfejs, poprzez który są wysyłane wychodzące datagramy rozsyłania grupowego.
- IP_MULTICAST_TTL: ustawia wartość Time To Live (TTL) w nagłówku IP wychodzących datagramów rozsyłania grupowego.
- IP_MULTICAST_LOOP: określa, czy kopia wychodzącego datagramu rozsyłania grupowego ma być dostarczana do hosta wysyłającego, o ile należy on do grupy rozsyłania grupowego.

Uwaga: Gniazda systemu i5/OS obsługują rozsyłanie grupowe IP dla rodziny adresów AF INET.



Przebieg zdarzeń w gnieździe: wysyłanie datagramów rozsyłania grupowego

Poniżej wyjaśniono sekwencję wywołań funkcji API gniazd, przedstawionych na rysunku. Opisano także relacje między dwiema aplikacjami, które wysyłają i odbierają datagramy rozsyłania grupowego. W pierwszym przykładzie jest używana następująca sekwencja wywołań funkcji API:

1. Funkcja API socket() zwraca deskryptor gniazda reprezentujący punkt końcowy. Instrukcja ta informuje również, że dla tego gniazda zostanie użyta rodzina adresów INET (Internet Protocol) i protokół transportowy TCP (SOCK_DGRAM). Gniazdo to wysyła 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 API setsockopt() ustawia opcję gniazda IP_MULTICAST_LOOP, aby system wysyłający nie otrzymywał wysłanych przezeń datagramów rozsyłania grupowego.
4. Funkcja API setsockopt() używa opcji gniazd IP_MULTICAST_IF, która definiuje interfejs lokalny, przez który będą przesyłane datagramy rozsyłania grupowego.
5. Funkcja API sendto() wysyła datagramy rozsyłania grupowego do podanych grupowych adresów IP.
6. Funkcja API close() zamyka wszelkie otwarte deskryptory gniazd.

Przebieg zdarzeń w gnieździe: odbieranie datagramów rozsyłania grupowego

W drugim przykładzie jest wykorzystywana następująca sekwencja wywołań funkcji API:

1. Funkcja API socket() zwraca deskryptor gniazda reprezentujący punkt końcowy. Instrukcja ta informuje również, że dla tego gniazda zostanie użyta rodzina adresów INET (Internet Protocol) i protokół transportowy TCP (SOCK_DGRAM). Gniazdo to wysyła datagramy do drugiej aplikacji.
2. Funkcja API setsockopt() ustawia opcję gniazd SO_REUSEADDR, umożliwiającą wielu aplikacjom odbieranie datagramów skierowanych do portu lokalnego o tym samym numerze.
3. Funkcja API 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 API setsockopt() używa opcji gniazd IP_ADD_MEMBERSHIP, łączącej grupę, dla której datagramy są przeznaczone. Dołączenie do grupy wiąże się z podaniem adresu grupy klasy D razem z adresem IP interfejsu lokalnego. System musi wywołać opcję gniazda IP_ADD_MEMBERSHIP dla każdego interfejsu lokalnego, 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 API read() odczytuje wysłane datagramy rozsyłania grupowego.
6. Funkcja API close() zamyka wszelkie otwarte deskryptory gniazd.

Pojęcia pokrewne

“Rozsyłanie grupowe IP” na stronie 60
Rozsyłanie grupowe IP pozwala aplikacjom na wysłanie pojedynczego datagramu IP, który zostanie odebrany przez grupę hostów w sieci.

Odsyłacze pokrewne

“Przykład: wysyłanie datagramów rozsyłania grupowego”
ten program przykładowy pozwala gniazdu na wysyłanie datagramów rozsyłania grupowego.

Informacje pokrewne

Funkcja API close() - zamykanie deskryptora gniazda lub pliku

Funkcja API socket() - tworzenie gniazd

Funkcja API bind() - ustawianie lokalnego adresu gniazda

Funkcja API setsockopt_ioctl() - ustawianie opcji gniazd

Funkcja API read() - odczytywanie danych z deskryptora

Funkcja API sendto() - wysyłanie danych

Przykład: wysyłanie datagramów rozsyłania grupowego

ten program przykładowy pozwala gniazdu na wysyłanie datagramów rozsyłania grupowego.

Uwaga: Korzystając z przykładów, użytkownik wyraża zgodę na warunki podane w sekcji “Licencja na kod oraz Informacje dotyczące kodu” na stronie 199.

```
#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) */
  /*                                                             */
  /* ------------------------------------------------------------*/

  /*
   * Utwórz gniazdo datagramowe, do którego ma być wysłany
   * datagram
   */
  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");
  }
}
```

Odsyłacze pokrewne

“Przykłady: rozsyłanie grupowe za pomocą rodziny adresów AF_INET” na stronie 172
Rozsyłanie grupowe IP pozwala aplikacjom na wysłanie pojedynczego datagramu IP, który zostanie odebrany przez grupę hostów w sieci.

Przykład: odbieranie datagramów rozsyłania grupowego

Ten program przykładowy pozwala gniazdu na odbieranie datagramów rozsyłania grupowego.

Uwaga: Korzystając z przykładów, użytkownik wyraża zgodę na warunki podane w sekcji “Licencja na kod oraz Informacje dotyczące kodu” na stronie 199.

```
#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: aktualizacja systemu DNS i wysyłanie do niego zapytań

W tym przykładzie przedstawiono sposób wysyłania zapytań do systemu nazw domen (DNS) oraz aktualizowania jego rekordów.

Uwaga: Korzystając z przykładów, użytkownik wyraża zgodę na warunki podane w sekcji “Licencja na kod oraz Informacje dotyczące kodu” na stronie 199.

```
/*********************************************************************/
/* Program ten aktualizuje DNS za pomocą sygnatury transakcji (TSIG)     */
/* służącej do podpisywania pakietów aktualizacyjnych. Następnie odpytuje */
/* serwer DNS, aby sprawdzić powodzenie aktualizacji.                    */
/*********************************************************************/

/*********************************************************************/
/* Pliki nagłówkowe wymagane przez program przykładowy                   */
/*********************************************************************/
#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 w 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);

   /* Umieść przetwarzanie w tym miejscu, aby móc sprawdzić wyniki      */
   /* i obsłużyć błędy                                                  */

   /* Budowanie buforu aktualizacji (pakietu do wysłania) na podstawie  */
   /* rekordów aktualizacji                                             */
   update_size = res_nmkupdate(&res, update_records,
                               update_buffer, buffer_length);

   /* Umieść przetwarzanie w tym miejscu, aby móc 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);

   /* Umieść przetwarzanie w tym miejscu, aby móc 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);

   /* Umieść przetwarzanie w tym miejscu, aby móc 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);

   /* Umieść przetwarzanie w tym miejscu, aby móc sprawdzić wyniki        */
   /* i obsłużyć błędy                                                    */
   }

   /*********************************************************************/
   /* Funkcje res_findzonecut(), res_nmkupdate() i res_nsendsigned()     */
   /* można zastąpić jednym wywołaniem funkcji 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 się nie powiodło.  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 się nie powiodło. Wynik = -1                     */
   /* errno: 0: Brak błędu.                                               */
   /* h_errno: 5: Nieznany host                                           */
   /*********************************************************************/
   return;
}
```

Pojęcia pokrewne

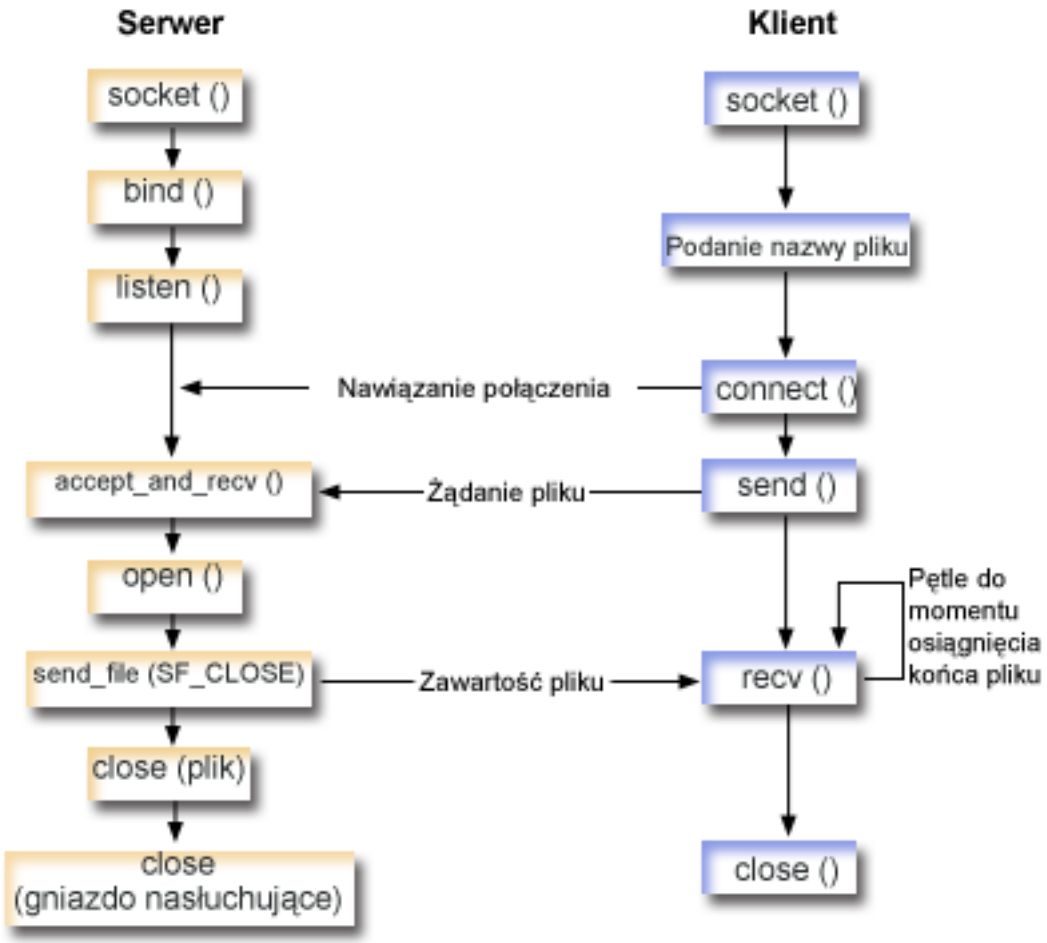
"Ochrona wątków" na stronie 57
Funkcja jest wątkowo bezpieczna (zapewnia ochronę wątków), jeśli w ramach tego samego procesu można ją uruchomić jednocześnie w wielu wątkach. Funkcję uznaje się za wątkowo bezpieczną tylko wtedy, gdy wszystkie funkcje przez nią wywoływane są również wątkowo bezpieczne. Funkcje API gniazd obejmują wątkowo bezpieczne funkcje systemowe i funkcje sieciowe.

Odsyłacze pokrewne

"Buforowanie danych" na stronie 66
Celem buforowania odpowiedzi na zapytania serwera DNS wykonywanego przez gniazda systemu i5/OS jest zmniejszenie ruchu w sieci. Pamięć podręczna jest dodawana i aktualizowana, gdy jest to wymagane.

Przykład: przesyłanie danych za pomocą funkcji API send_file() i accept_and_recv()

Poniższe programy przykładowe pozwalają serwerowi na komunikowanie się z klientem za pomocą funkcji API send_file() i accept_and_recv().



Przebieg zdarzeń w gnieździe: wysyłanie zawartości pliku przez serwer

Poniżej wyjaśniono sekwencję wywołań funkcji API gniazd, przedstawionych na rysunku. Opisano także relacje między dwiema aplikacjami, które wysyłają i odbierają pliki. W pierwszym przykładzie jest używana następująca sekwencja wywołań funkcji API:

1. Serwer wywołuje funkcje socket(), bind() i listen() w celu utworzenia gniazda nasłuchującego.

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 API 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 API memset() zeruje wszystkie pola struktury sf_parms. Serwer ustawia 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, nie trzeba więc podawać wartości pola pozycji w pliku.
6. Serwer przesyła zawartość pliku za pomocą funkcji API send_file(). Funkcja API send_file() nie kończy działania, dopóki cały plik nie zostanie wysłany lub funkcja nie zostanie przerwana. Funkcja API send_file() jest bardziej wydajna, ponieważ aplikacja nie musi wchodzić w pętlę funkcji read() i send(), dopóki transfer pliku nie zostanie zakończony.
7. Serwer podaje opcję SF_CLOSE dla funkcji API send_file(). Opcja SF_CLOSE informuje funkcję API send_file() o konieczności automatycznego zamknięcia połączenia gniazda z chwilą pomyślnego przesłania ostatniego bajtu pliku i buforu końcowego (jeśli został podany). Aplikacja nie musi wywoływać funkcji close(), jeśli została podana opcja SF_CLOSE.

Przebieg zdarzeń w gnieździe: żądanie pliku przez klienta

W drugim przykładzie jest wykorzystywana następująca sekwencja wywołań funkcji API:

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. Klient wywołuje funkcję API socket() w celu utworzenia deskryptora gniazda.
3. Klient wywołuje funkcję API connect() w celu nawiązania połączenia z serwerem. Adres IP serwera uzyskano w pierwszym kroku.
4. Klient wywołuje funkcję API 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.

   Pojęcia pokrewne

   “Przesyłanie danych pliku - funkcje send_file() i accept_and_recv()” na stronie 61
   Obsługa gniazd w systemie i5/OS obejmuje funkcje API send_file() i accept_and_recv() umożliwiające szybsze i łatwiejsze przesyłanie danych między połączonymi gniazdami.

   Informacje pokrewne

   Funkcja API accept() - oczekiwanie na żądanie i nawiązywanie połączenia

   Funkcja API recv() - odbieranie danych

   Funkcja API send() - wysyłanie danych

   Funkcja API listen() - nasłuchiwanie przychodzących żądań połączenia

   Funkcja API close() - zamykanie deskryptora gniazda lub pliku

   Funkcja API socket() - tworzenie gniazd

   Funkcja API bind() - ustawianie lokalnego adresu gniazda

   Funkcja API getsockname() - wczytywanie lokalnego adresu gniazda

Funkcja API open() - otwieranie pliku

Funkcja API read() - odczytywanie danych z deskryptora

Funkcja API connect() - nawiązywanie połączenia lub ustanawianie adresu docelowego

Przykład: użycie funkcji API accept_and_recv() i send_file() w celu wysłania zawartości pliku

Poniższe przykłady umożliwiają serwerowi komunikację z klientem za pomocą funkcji send_file() i accept_and_recv().

Uwaga: Korzystając z przykładów, użytkownik wyraża zgodę na warunki podane w sekcji “Licencja na kod oraz Informacje dotyczące kodu” na stronie 199.

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

   /*********************************************/
   /* Ustaw bit SO_REUSEADDR, aby nie trzeba było   */
   /* czekać 2 minut zanim serwer zostanie ponownie */
   /* uruchomiony                                   */
   /*********************************************/
```

```
rc = setsockopt(listen_sd,
                SOL_SOCKET,
                SO_REUSEADDR,
                (char *)&flag,
                sizeof(flag));
if (rc < 0)
{
   perror("Niepowodzenie funkcji 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 funkcji bind()");
   close(listen_sd);
   exit(-1);
}

/*************************************************/
/* Ustaw parametr backlog funkcji listen         */
/*************************************************/
rc = listen(listen_sd, 5);
if (rc < 0)
{
   perror("Niepowodzenie funkcji 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++)
{
   /**********************************************/
   /* Czekaj 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 funkcji 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 funkcji 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 funkcji 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);
}
```

Informacje pokrewne

Funkcja API send_file() - wysyłanie pliku za pomocą połączenia przez gniazdo

Funkcja API accept_and_recv()

Przykład: klient żądający pliku

Poniższy program przykładowy pozwala klientowi na zażądanie pliku z serwera i zaczekanie, aż serwer odeśle w odpowiedzi zawartość pliku.

Uwaga: Korzystając z przykładów, użytkownik wyraża zgodę na warunki podane w sekcji “Licencja na kod oraz Informacje dotyczące kodu” na stronie 199.

```
/*********************************************/
/* 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;

   /*********************************************/
   /* Zainicjuj strukturę 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 strumieniowe AF_INET          */
/************************************************/
sockfd = socket(AF_INET, SOCK_STREAM, 0);
if (sockfd < 0)
{
   perror("Niepowodzenie funkcji socket()");
   exit(-1);
}
printf("Gniazdo zakończyło działanie.\n");

/************************************************/
/* Połącz się z serwerem                        */
/************************************************/
rc = connect(sockfd,
             (struct sockaddr *)&addr,
             sizeof(struct sockaddr_in));
if (rc < 0)
{
   perror("Niepowodzenie funkcji 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 funkcji 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 funkcji 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 to jedno z wielu narzędzi dostarczanych wraz z produktem System i. Wszystkie narzędzia znajdują się w bibliotece QUSRTOOL. Narzędzie Xsockets umożliwia programistom interaktywną pracę z funkcjami API gniazd.

Narzędzie Xsockets umożliwia realizację następujących zadań:

- zapoznanie się z funkcjami API gniazd,
- interaktywne wykonywanie konkretnych scenariuszy, co ułatwia debugowanie.

Uwaga: Narzędzie Xsockets jest udostępniane w postaci takiej jaka jest (as-is).

Wymagania wstępne dla narzędzia Xsockets

Przed rozpoczęciem pracy z narzędziem Xsockets należy wykonać następujące czynności:

- Zainstaluj język ILE C.
- Zainstaluj element System Openness Includes (opcja 13) licencjonowanego programu IBM i5/OS.
- Zainstaluj licencjonowany program IBM HTTP Server for i5/OS (5761-DG1).

  Uwaga: Program ten jest potrzebny do korzystania z narzędzia Xsockets za pośrednictwem przeglądarki WWW.
- Zainstalowany musi być także licencjonowany program IBM Developer Kit for Java (5761-JV1).

  Uwaga: Program ten jest potrzebny do korzystania z narzędzia Xsockets za pośrednictwem przeglądarki WWW.

Konfigurowanie narzędzia Xsockets

Narzędzie Xsockets jest dostępne w dwóch wersjach. Wersja pierwsza jest zintegrowana z klientem platformy System i. Jest ona w całości tworzona za pomocą pierwszego zestawu instrukcji. Druga wersja używa jako klienta przeglądarki WWW.

Aby móc korzystać z przeglądarki WWW jako klienta, należy najpierw wykonać czynności związane z konfigurowaniem wersji zintegrowanej.

Aby utworzyć narzędzie Xsockets, należy wykonać następujące czynności:

1. Aby rozpakować narzędzie, wpisz:

   ```
   CALL QUSRTOOL/UNPACKAGE ('*ALL      ' 1)
   ```

   w wierszu komend.

Uwaga: Między apostrofem (') otwierającym a zamykającym musi się znajdować 10 znaków.

2. Aby do listy bibliotek dodać bibliotekę QUSRTOOL, wpisz:
   ```
   ADDLIBLE QUSRTOOL
   ```

   w wierszu komend.
3. Aby utworzyć bibliotekę, w której zostaną utworzone pliki programu Xsockets, wpisz:
   ```
   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.

   Uwaga: W przypadku używania biblioteki XSOCKETS można pominąć jeden z etapów konfiguracji narzędzia Xsockets do pracy w sieci WWW. Obiektów narzędzia Xsocket nie należy dodawać do biblioteki QUSRTOOL. Dodanie obiektów Xsockets do biblioteki QUSRTOOL może zakłócić działanie innych narzędzi w tym katalogu.
4. Aby dodać tę bibliotekę do listy bibliotek, w wierszu komend wpisz: ADDLIBLE <*nazwa-biblioteki*>. Parametr <*nazwa-biblioteki*> należy zastąpić nazwą biblioteki utworzonej w punkcie 3. Jeśli na przykład utworzono bibliotekę o nazwie MYXSOCKET, należy wpisać: ADDLIBLE MYXSOCKET.
5. Aby utworzyć program instalacyjny TSOCRT, który automatycznie zainstaluje narzędzie Xsockets, wpisz: CRTCLPGM <*nazwa-biblioteki*>/TSOCRT QUSRTOOL/QATTCL w wierszu komend.
6. Aby wywołać program instalacyjny, w wierszu komend wpisz:
   ```
   CALL TSOCRT nazwa-biblioteki.
   ```

   Zamiast frazy nazwa-biblioteki użyj nazwy biblioteki utworzonej w punkcie 3. Aby na przykład utworzyć narzędzie w bibliotece MYXSOCKET, wpisz:
   ```
   CALL TSOCRT MYXSOCKET.
   ```

   Uwaga: Może to potrwać kilka minut.

Jeśli użytkownik uruchamiający program TSOCRT nie ma uprawnienia specjalnego do zarządzania zadaniem (*JOBCTL), to podczas próby przekazania deskryptora do zadania innego niż właśnie uruchomione funkcja gniazda givedescriptor() zwróci błędy.

Przez program TSOCRT są tworzone: program CL, program ILE C (dwa moduły), dwa programy usługowe ILE C (dwa moduły) oraz trzy zbiory ekranowe. Bibliotekę należy każdorazowo dodawać do listy bibliotek przed uruchomieniem narzędzia Xsockets. Wszystkie obiekty utworzone za pomocą narzędzia będą miały nazwę z przedrostkiem TSO.

Uwaga: Zintegrowana wersja narzędzia nie obsługuje funkcji API GSKit gniazd chronionych. Aby napisać program obsługi gniazd korzystający z tych funkcji, należy użyć wersji opartej na przeglądarce WWW.

Pojęcia pokrewne

"Korzystanie z narzędzia Xsockets" na stronie 195
Z narzędzia Xsockets można korzystać przy użyciu zintegrowanego klienta albo za pomocą przeglądarki WWW.

Zadania pokrewne

"Korzystanie ze zintegrowanego narzędzia Xsockets" na stronie 195
Aby używać narzędzia Xsockets na zintegrowanym kliencie, należy wykonać następujące czynności.

"Aktualizowanie plików konfiguracyjnych" na stronie 192
Po zainstalowaniu zintegrowanego narzędzia Xsockets w kilku plikach konfiguracyjnych jego instancji należy ręcznie wprowadzić zmiany.

“Korzystanie z narzędzia Xsockets w przeglądarce WWW” na stronie 196
Aby korzystać z narzędzia Xsockets w przeglądarce WWW, należy wykonać następujące czynności.

Co tworzy zintegrowana konfiguracja Xsocket

W poniższej tabeli zawarto listę obiektów tworzonych przez program instalacyjny. Wszystkie tak utworzone obiekty rezydują w podanej bibliotece.

Tabela 18. Obiekty utworzone podczas instalacji narzędzia Xsocket

Nazwa obiektu	Nazwa podzbioru	Nazwa zbioru źródłowego	Typ obiektu	Rozszerzenie	Opis
TSOJNI	TSOJNI	QATTSYSC	*MODULE	C	Moduł stanowiący interfejs między 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 używany do obsługi interaktywnego narzędzia SOCKETS.
TSOJNI	nie dotyczy	nie dotyczy	*SRVPGM	C	Program serwisowy stanowiący interfejs między JSP a TSOSTSOC, używany 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 procedury wywołujące funkcje gniazd.

Tabela 18. Obiekty utworzone podczas instalacji narzędzia Xsocket (kontynuacja)

Nazwa obiektu	Nazwa podzbioru	Nazwa zbioru źródłowego	Typ obiektu	Rozszerzenie	Opis
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, identyfikatora użytkownika i numeru zadania.
TSODSP	TDSPDSP	QATTDDS	*FILE	DSPF	Zbiór ekranowy używany przez narzędzie Xsockets do tworzenia okna głównego 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.
QATTIFS2	nie dotyczy	nie dotyczy	*FILE	PF-DTA	Zawiera zbiór JAR używany przez uproszczoną infrastrukturę WWW (Lightweight Web Infrastructure - LWI).

Konfigurowanie narzędzia Xsockets do korzystania z przeglądarki WWW

Narzędzie Xsockets można skonfigurować w taki sposób, aby było dostępne za pośrednictwem przeglądarki WWW. Aby utworzyć wiele instancji serwera, można zastosować te instrukcje wielokrotnie w tym samym systemie. Wiele instancji pozwala na jednoczesne uruchamianie wielu wersji na różnych portach nasłuchujących.

Pojęcia pokrewne

"Korzystanie z narzędzia Xsockets" na stronie 195
Z narzędzia Xsockets można korzystać przy użyciu zintegrowanego klienta albo za pomocą przeglądarki WWW.

Zadania pokrewne

"Korzystanie z narzędzia Xsockets w przeglądarce WWW" na stronie 196
Aby korzystać z narzędzia Xsockets w przeglądarce WWW, należy wykonać następujące czynności.

Konfigurowanie zintegrowanego serwera aplikacji WWW

Aby można było korzystać z narzędzia Xsockets w przeglądarce WWW, należy skonfigurować zintegrowany serwer aplikacji WWW.

Przed skonfigurowaniem przeglądarki WWW do pracy z narzędziem Xsockets, należy najpierw skonfigurować to narzędzie. Informacje na ten temat znajdują się w sekcji Konfigurowanie narzędzia Xsockets.

1. Sprawdź, czy w podsystemie QHTTPSVR działa instancja administrująca serwerem HTTP. Jeśli nie, to możesz ją uruchomić za pomocą następującej komendy CL:
   ```
   STRTCPSVR SERVER(*HTTP) HTTPSVR(*ADMIN)
   ```
2. W przeglądarce WWW wpisz:
   ```
   http://<nazwa_systemu>:2001/
   ```
   gdzie <*nazwa_systemu*> to nazwa komputera, na którym działa system. Na przykład: http://mysystemi:2001/.
3. Na stronie Zadania systemu i5/OS wybierz zakładkę **Zarządzanie siecią IBM i5/OS**.
4. Z górnego menu wybierz zakładkę **Konfiguracja**.
5. Wybierz opcję **Tworzenie serwera aplikacji** (Create Application Server).
6. W sekcji Zintegrowany serwer aplikacji WWW wybierz opcję **V7.1** i kliknij przycisk **Dalej**.
7. Wpisz nazwę instancji serwera i kliknij przycisk **Dalej**. Jeśli na przykład instancja ta obsługuje narzędzie Xsockets w przeglądarce, możesz użyć nazwy xsocket. Obok zintegrowanego serwera aplikacji WWW zostanie utworzony nowy serwer HTTP (oparty na oprogramowaniu Apache). .

   Uwaga: Użyj domyślnej nazwy i opisu serwera HTTP.
8. Wybierz adres IP oraz dostępny port, z którego chcesz korzystać, po czym kliknij przycisk **Dalej**.

   Uwaga: Użyj portu o numerze większym niż 1024. Nie należy wybierać domyślnego portu o numerze 80.
9. Wybierz zakres portów wewnętrznych, z których będzie korzystać serwer aplikacji, po czym kliknij przycisk **Dalej**.

   Uwaga: Użyj portu o numerze większym niż 1024.
10. Kliknij przycisk **Dalej**, aby użyć wartości domyślnej przy określaniu identyfikatora użytkownika.
11. Na ekranie Przykładowe aplikacje (Sample Applications) kliknij przycisk **Dalej**.
12. Kliknij przycisk **Zakończ** (Finish), aby potwierdzić ustawienia konfiguracyjne serwera aplikacji i serwera HTTP (opartego na oprogramowaniu Apache).

Zadania pokrewne

“Aktualizowanie plików konfiguracyjnych”
Po zainstalowaniu zintegrowanego narzędzia Xsockets w kilku plikach konfiguracyjnych jego instancji należy ręcznie wprowadzić zmiany.

“Testowanie narzędzia Xsockets w przeglądarce WWW” na stronie 194
Po zakończeniu konfigurowania aplikacji WWW dla narzędzia Xsockets można przetestować to narzędzie w przeglądarce. Instancje serwera i aplikacji powinny już być uruchomione.

“Korzystanie z narzędzia Xsockets w przeglądarce WWW” na stronie 196
Aby korzystać z narzędzia Xsockets w przeglądarce WWW, należy wykonać następujące czynności.

Aktualizowanie plików konfiguracyjnych

Po zainstalowaniu zintegrowanego narzędzia Xsockets w kilku plikach konfiguracyjnych jego instancji należy ręcznie wprowadzić zmiany.

Aktualizacji wymagają następujące pliki: JAR, web.xml oraz httpd.conf.

1. Kopiowanie pliku JAR W wierszu komend wpisz komendę:
   ```
   CPY OBJ('/QSYS.LIB/XXXX.LIB/QATTIFS2.FILE/XSOCK.MBR')
      TOOBJ('/www/<nazwa_serwera>/xsock.jar') FROMCCSID(*OBJ)
   TOCCSID(819) OWNER(*NEW)
   ```

gdzie *XXXX* to nazwa biblioteki utworzonej podczas konfigurowania narzędzia Xsockets, a <*nazwa_serwera*> to nazwa instancji serwera utworzonej podczas konfigurowania serwera Apache. Jest to katalog zintegrowanego systemu plików, w którym ma być przechowywany plik JAR narzędzia XSockets.

2. Opcjonalne: Zaktualizuj plik web.xml.

   Uwaga: Tę czynność należy wykonać tylko wtedy, gdy podczas konfigurowania narzędzia Xsockets zainstalowano je w bibliotece innej niż XSOCKETS.

   a. W wierszu komend wpisz:
   ```
   CD DIR('/www/<nazwa_serwera>')
   ```
   gdzie <*nazwa_serwera*> to nazwa instancji serwera utworzonej podczas konfigurowania oprogramowania Apache.

   b. W wierszu komend wpisz:
   ```
   STRQSH CMD('jar xf xsock.jar').
   ```
   Spowoduje to wyodrębnienie plików konfiguracyjnych zapisanych w pliku JAR narzędzia XSockets.

   c. W wierszu komend wpisz:
   ```
   wrklnk 'com.ibm.i5OS.xSockets/WEB-INF/web.xml'
   ```
   d. Kliknij przycisk F2 (Edycja), aby zmienić ten plik.

   e. Znajdź wiersz </servlet-class> w pliku web.xml.

   f. Zaktualizuj kod za tym wierszem:
   ```
   <init-param>
             <param-name>library</param-name>
             <param-value>xsockets</param-value>
   </init-param>
   ```
   Zamiast *xsockets* wstaw nazwę biblioteki utworzonej podczas konfiguracji narzędzia Xsockets.

   g. Zapisz plik i zakończ sesję edycji.

   h. W wierszu komend wpisz:
   ```
   STRQSH CMD('jar cf xsock.jar com.ibm.i5OS.xSockets').
   ```
   Spowoduje to utworzenie nowego pliku JAR narzędzia XSockets zawierającego zaktualizowane pliki konfiguracyjne.

3. Opcjonalne: Dodaj sprawdzanie uprawnień do pliku httpd.conf. Wymusza to uwierzytelnianie przez serwer Apache użytkowników próbujących skorzystać z aplikacji Xsockets.

   Uwaga: Podczas tworzenia gniazd UNIX niezbędne są także uprawnienia do zapisu.

   a. W wierszu komend wpisz:
   ```
   wrklnk '/www/<nazwa_serwera>/conf/httpd.conf'
   ```
   gdzie <*nazwa_serwera*> to nazwa instancji serwera utworzonej podczas konfigurowania serwera Apache. Jeśli na przykład nazwą serwera jest xsocks, wpisz:
   ```
   wrklnk '/www/xsocks/conf/httpd.conf'
   ```
   b. Kliknij przycisk F2 (Edycja), aby zmienić ten plik.

   c. Na końcu pliku wstaw poniższe wiersze.
   ```
   <Lokalizacja /xsock>
      AuthName "X Socket"
      AuthType Basic
      PasswdFile %%SYSTEM%%
      UserId %%CLIENT%%
   ```

```
    Require valid-user
    order allow,deny
    allow from all
</Lokalizacja>
```

d. Zapisz plik i zakończ sesję edycji.

Zadania pokrewne

“Konfigurowanie zintegrowanego serwera aplikacji WWW” na stronie 191
Aby można było korzystać z narzędzia Xsockets w przeglądarce WWW, należy skonfigurować zintegrowany serwer aplikacji WWW.

“Konfigurowanie narzędzia Xsockets” na stronie 188
Narzędzie Xsockets jest dostępne w dwóch wersjach. Wersja pierwsza jest zintegrowana z klientem platformy System i. Jest ona w całości tworzona za pomocą pierwszego zestawu instrukcji. Druga wersja używa jako klienta przeglądarki WWW.

“Konfigurowanie aplikacji WWW Xsockets”
Aby można było skorzystać z narzędzia Xsockets w przeglądarce WWW, należy najpierw skonfigurować zintegrowany serwer aplikacji WWW oraz instancję serwera HTTP (opartego na oprogramowaniu Apache). Po wykonaniu tego zadania należy skonfigurować nową aplikację, która umożliwi skorzystanie z narzędzia Xsockets.

Konfigurowanie aplikacji WWW Xsockets

Aby można było skorzystać z narzędzia Xsockets w przeglądarce WWW, należy najpierw skonfigurować zintegrowany serwer aplikacji WWW oraz instancję serwera HTTP (opartego na oprogramowaniu Apache). Po wykonaniu tego zadania należy skonfigurować nową aplikację, która umożliwi skorzystanie z narzędzia Xsockets.

1. W sekcji **Zarządzaj** (Manage) wybierz utworzony przez siebie serwer aplikacji.
2. W sekcji **Kreatory serwera aplikacji** (Application Server Wizards) wybierz w lewym panelu opcję **Instalowanie nowej aplikacji** (Install New Application).
3. Określ położenie pliku JAR, który zawiera tę aplikację, po czym kliknij przycisk **Dalej**. Chodzi tu o plik JAR utworzony z biblioteki ’/QSYS.LIB/XXX.LIB/QATTIFS2.FILE/XSOCK.MBR’ i zaktualizowany podczas aktualizacji plików konfiguracyjnych.
4. Wpisz nazwę aplikacji i kliknij przycisk **Dalej**. Jeśli na przykład aplikacja ta obsługuje narzędzie Xsockets w przeglądarce, możesz użyć nazwy XSockets.
5. Kliknij przycisk **Dalej**, aby zaakceptować wartości domyślne na stronie **Odwzorowanie portów kontekstowego katalogu głównego** (Context Root Port Mapping).
6. Kliknij przycisk **Zakończ**, aby zakończyć konfigurowanie aplikacji dla narzędzia Xsockets.

Zadania pokrewne

“Aktualizowanie plików konfiguracyjnych” na stronie 192
Po zainstalowaniu zintegrowanego narzędzia Xsockets w kilku plikach konfiguracyjnych jego instancji należy ręcznie wprowadzić zmiany.

Testowanie narzędzia Xsockets w przeglądarce WWW

Po zakończeniu konfigurowania aplikacji WWW dla narzędzia Xsockets można przetestować to narzędzie w przeglądarce. Instancje serwera i aplikacji powinny już być uruchomione.

1. Jeśli instancje serwera i aplikacji nie są uruchomione, to w celu uruchomienia instancji serwera wpisz w wierszu komend następującą komendę:

   ```
   STRTCPSVR SERVER(*HTTP) HTTPSVR(<nazwa_serwera>),
   ```

   gdzie <*nazwa_serwera*> to nazwa instancji serwera utworzonej podczas konfigurowania serwera HTTP (opartego na oprogramowaniu Apache). Może to zająć trochę czasu.
2. Sprawdź status serwera za pomocą komendy Praca z zadaniami aktywnymi (Work with Active Jobs - WRKACTJOB). Powinno zostać wyświetlone jedno zadanie o nazwie *nazwa_serwera*, funkcja PGM-QLWISVR ze statusem JVAW, a wszelkie pozostałe zadania powinny mieć status SIGW. Jeśli tak jest, można przejść do kolejnego punktu.

3. W przeglądarce WWW wpisz następujący adres URL:

   ```
   http://<nazwa systemu>:<port>/xsock/index
   ```

   gdzie <*nazwa_systemu*> to nazwa komputera, na którym działa system; <*port*> to numer portu wybrany podczas konfigurowania oprogramowania Apache.
4. Po pojawieniu się podpowiedzi wpisz nazwę użytkownika oraz hasło do serwera. Powinien zostać wyświetlony klient narzędzia Xsockets.

Zadania pokrewne

"Konfigurowanie zintegrowanego serwera aplikacji WWW" na stronie 191
Aby można było korzystać z narzędzia Xsockets w przeglądarce WWW, należy skonfigurować zintegrowany serwer aplikacji WWW.

Korzystanie z narzędzia Xsockets

Z narzędzia Xsockets można korzystać przy użyciu zintegrowanego klienta albo za pomocą przeglądarki WWW.

Aby pracować ze zintegrowaną wersją narzędzia Xsockets, należy je skonfigurować. Jeśli użytkownik preferuje pracę z narzędziem w środowisku przeglądarki, to oprócz skonfigurowania narzędzia Xsockets dla zintegrowanego klienta należy również wykonać czynności opisane w sekcji Konfigurowanie narzędzia Xsockets do korzystania z przeglądarki WWW. Wiele pojęć jest podobnych w obu wersjach narzędzia. Obydwa narzędzia umożliwiają interaktywne wywoływanie funkcji API gniazd i dostarczają informacji o kodach błędów zwracanych przez te funkcje. Występują jednak pewne różnice w ich interfejsach.

Uwaga: Aby pracować z programami obsługi gniazd korzystającymi z funkcji API GSKit gniazd chronionych, należy użyć wersji opartej na przeglądarce WWW.

Pojęcia pokrewne

"Konfigurowanie narzędzia Xsockets do korzystania z przeglądarki WWW" na stronie 191
Narzędzie Xsockets można skonfigurować w taki sposób, aby było dostępne za pośrednictwem przeglądarki WWW. Aby utworzyć wiele instancji serwera, można zastosować te instrukcje wielokrotnie w tym samym systemie. Wiele instancji pozwala na jednoczesne uruchamianie wielu wersji na różnych portach nasłuchujących.

Zadania pokrewne

"Konfigurowanie narzędzia Xsockets" na stronie 188
Narzędzie Xsockets jest dostępne w dwóch wersjach. Wersja pierwsza jest zintegrowana z klientem platformy System i. Jest ona w całości tworzona za pomocą pierwszego zestawu instrukcji. Druga wersja używa jako klienta przeglądarki WWW.

Korzystanie ze zintegrowanego narzędzia Xsockets

Aby używać narzędzia Xsockets na zintegrowanym kliencie, należy wykonać następujące czynności.

1. Do listy bibliotek systemu dodaj bibliotekę, która zawiera narzędzie Xsockets; w tym celu w wierszu komend wpisz następującą komendę:

   ```
   ADDLIBLE <nazwa-biblioteki>
   ```

   gdzie <nazwa-biblioteki> jest nazwą biblioteki utworzonej podczas konfigurowania zintegrowanego klienta Xsockets. Jeśli na przykład nazwą biblioteki jest MYXSOCKET, wpisz:

   ```
   ADDLIBLE MYXSOCKET
   ```
2. W wierszu komend wpisz:

   ```
   CALL TSOXSOCK
   ```
3. Zostanie wyświetlone okno Xsockets z paskiem menu i polami wyboru, które umożliwiają dostęp do wszystkich procedur gniazd. Okno to jest wyświetlane zawsze po wybraniu funkcji API gniazd. Interfejsu tego można używać do wybierania istniejących programów używających gniazd. Aby utworzyć nowe gniazdo, wykonaj poniższe czynności:
   a. Z listy funkcji API gniazd 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 opcję **Deskryptor** (Descriptor), a następnie opcję **Wybierz deskryptor** (Select descriptor).

   Uwaga: Jeśli istnieją już inne deskryptory gniazd, to zostanie wyświetlona lista aktywnych deskryptorów gniazd.
d. Z wyświetlonej listy wybierz utworzony przez siebie deskryptor gniazda.

   Uwaga: Jeśli istnieją inne deskryptory gniazd, narzędzie automatycznie zastosuje tę funkcję API do najnowszego deskryptora gniazda.

4. Z listy funkcji API gniazd wybierz tę, z którą chcesz pracować. Dla wybranej funkcji API zostanie użyty deskryptor gniazda wybrany w punkcie 3c. Po wybraniu funkcji API gniazd na ekranie zostanie wyświetlona grupa okien, w których można wprowadzić szczegółowe informacje na temat tej funkcji API. Jeśli na przykład zostanie wybrana funkcja connect(), w wyświetlonych oknach trzeba będzie podać długość adresu, rodzinę adresów i dane adresu. Wybrana funkcja API gniazd zostanie następnie wywołana z podanymi informacjami. Jeśli podczas wywoływania funkcji API gniazda wystąpią błędy, to zostaną one wyświetlone w postaci kodów errno.

Uwagi:

1. Narzędzie Xsockets używa graficznej obsługi usług DDS. Dlatego też sposób wprowadzania danych i dokonywania wyborów oraz to, jakie okna są widoczne, zależy od tego, czy używa się terminalu graficznego, czy tekstowego. Na przykład na terminalu graficznym będzie widoczne pole wyboru funkcji gniazda; na ekranie tekstowym będzie to zwykłe, pojedyncze pole.
2. Należy wiedzieć, że w gnieździe są dostępne żądania ioctl(), które nie zostały zaimplementowane w narzędziu.

Korzystanie z narzędzia Xsockets w przeglądarce WWW

Aby korzystać z narzędzia Xsockets w przeglądarce WWW, należy wykonać następujące czynności.

Zanim przystąpisz do używania narzędzia Xsockets w przeglądarce WWW, upewnij się, że wykonałeś wszystkie czynności niezbędne do skonfigurowania narzędzia Xsockets oraz przeglądarki WWW. Sprawdź także, czy jest włączona obsługa informacji cookie.

1. W przeglądarce WWW wpisz:

   ```
   http://nazwa_systemu:2001/
   ```

   gdzie *nazwa_systemu* jest nazwą systemu zawierającego instancję serwera.
2. Wybierz opcję **Administration** (Administrowanie).
3. Z lewego paska nawigacyjnego wybierz opcję **Manage HTTP Servers** (Zarządzanie serwerami HTTP).
4. Wybierz nazwę instancji i kliknij przycisk **Start**. Możesz także uruchomić instancję serwera z wiersza komend; w tym celu wpisz:

   ```
   STRTCPSVR SERVER(*HTTP) HTTPSVR(<nazwa_instancji>)
   ```

   gdzie <*nazwa_instancji*> to nazwa serwera HTTP utworzonego podczas konfigurowania serwera Apache. Można na przykład użyć instancji serwera o nazwie xsocks.
5. Aby uzyskać dostęp do aplikacji Xsockets poprzez sieć WWW, w przeglądarce wpisz poniższy adres URL:

   ```
   http://<nazwa systemu>:<port>/xsock/index
   ```

   gdzie <*nazwa_systemu*> to nazwa komputera, na którym działa system; <port> to numer portu podany podczas tworzenia instancji serwera HTTP. Jeśli na przykład nazwą systemu jest mySystemi, a instancja serwera HTTP nasłuchuje na porcie 1025, należy wpisać:

   ```
   http://mySystemi:1025/xsock/index
   ```
6. Po załadowaniu narzędzia Xsockets w przeglądarce WWW można pracować z istniejącym deskryptorem gniazd lub utworzyć nowy. Aby utworzyć nowy deskryptor gniazd, wykonaj poniższe czynności:

a. W menu **Xsocket** wybierz opcję **gniazdo**.
b. W wyświetlonym oknie **Xsocket Query** wybierz odpowiednią rodzinę adresów, typ gniazda i protokół dla tego deskryptora gniazda. Kliknij przycisk **Wyślij**. Po ponownym załadowaniu strony nowy deskryptor gniazda zostanie wyświetlony w menu rozwijanym **Socket** (Gniazdo).
c. Z menu **Xsocket** wybierz wywołania funkcji API, które chcesz zastosować do tego deskryptora gniazda. Podobnie jak w zintegrowanej wersji Xsockets, jeśli nie zostanie wybrany deskryptor gniazda, to wywołania funkcji będą automatycznie zastosowane do najnowszego deskryptora gniazda.

Pojęcia pokrewne

"Konfigurowanie narzędzia Xsockets do korzystania z przeglądarki WWW" na stronie 191
Narzędzie Xsockets można skonfigurować w taki sposób, aby było dostępne za pośrednictwem przeglądarki WWW. Aby utworzyć wiele instancji serwera, można zastosować te instrukcje wielokrotnie w tym samym systemie. Wiele instancji pozwala na jednoczesne uruchamianie wielu wersji na różnych portach nasłuchujących.

Zadania pokrewne

"Konfigurowanie narzędzia Xsockets" na stronie 188
Narzędzie Xsockets jest dostępne w dwóch wersjach. Wersja pierwsza jest zintegrowana z klientem platformy System i. Jest ona w całości tworzona za pomocą pierwszego zestawu instrukcji. Druga wersja używa jako klienta przeglądarki WWW.

"Konfigurowanie zintegrowanego serwera aplikacji WWW" na stronie 191
Aby można było korzystać z narzędzia Xsockets w przeglądarce WWW, należy skonfigurować zintegrowany serwer aplikacji WWW.

Usuwanie obiektów utworzonych za pomocą narzędzia Xsockets

Niekiedy konieczne jest usunięcie obiektów, które zostały utworzone za pomocą narzędzia Xsockets. W trakcie działania programu instalacyjnego tworzony jest program o nazwie TSODLT. Służy do usuwania obiektów utworzonych przez narzędzie Xsockets (z wyjątkiem biblioteki i samego programu TSODLT) i/lub wykorzystywanych przez Xsockets podzbiorów źródłowych.

Usuwanie obiektów jest możliwe dzięki następującemu zestawowi komend:

Aby usunąć TYLKO podzbiory źródłowe używane przez narzędzie, wpisz:

```
CALL TSODLT   (*YES *NONE)
```

Aby usunąć TYLKO obiekty utworzone za pomocą narzędzia, wpisz:

```
CALL TSODLT   (*NO  nazwa-biblioteki)
```

Aby usunąć ZARÓWNO podzbiory źródłowe, jak i obiekty utworzone za pomocą narzędzia, wpisz:

```
CALL TSODLT   (*YES nazwa-biblioteki)
```

Dostosowywanie narzędzia Xsockets

Narzędzie Xsockets można dostosować poprzez wprowadzenie dodatkowej obsługi sieciowych procedur gniazd (na przykład inet_addr()).

W przypadku dostosowywania narzędzia do własnych potrzeb nie zaleca się dokonywania zmian w bibliotece QUSRTOOL. Należy natomiast skopiować zbiory źródłowe do osobnej biblioteki i w niej 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, zmian należy dokonać również w programie TSOCRT). Do usunięcia starych wersji obiektów narzędzia przed utworzeniem nowej wersji służy program TSODLT.

Narzędzia do serwisowania

W związku z ciągłym wzrostem liczby zastosowań gniazd i gniazd chronionych w aplikacjach i serwerach używanych w e-biznesie muszą się również odpowiednio rozwijać narzędzia serwisowe.

Udoskonalone narzędzia serwisowe ułatwiają śledzenie działania programów używających gniazd w celu znajdowania i usuwania błędów w aplikacjach używających gniazd i SSL. Narzędzia te umożliwiają programistom i pracownikom centrum obsługi umiejscowienie problemów związanych z użyciem gniazd poprzez wybór takich parametrów gniazd, jak adres IP lub informacje o porcie.

W poniższej tabeli zawarto przegląd informacji o tych narzędziach serwisowych.

Tabela 19. Narzędzia serwisowe dla gniazd i gniazd chronionych

Narzędzie serwisowe	**Opis**
Filtrowanie śledzenia Licencjonowanego Kodu Wewnętrznego (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 tylko do tych gniazd, które mają ustawioną opcję SO_DEBUG. Od wersji V5R2 śledzenie Licencjonowanego Kodu Wewnętrznego można filtrować według wątku, zadania, profilu użytkownika, nazwy zadania i nazwy serwera.
Śledzenie zadania za pomocą komendy STRTRC SSNID(*GEN) JOBTRCTYPE(*TRCTYPE) TRCTYPE((*SOCKETS *ERROR))	Komenda STRTRC udostępnia dodatkowe parametry, dzięki którym dane wyjściowe są oddzielone od punktów śledzenia niezwiązanych z gniazdami. Dane te zawierają kod powrotu i kod błędu w przypadku wystąpienia błędu podczas operacji gniazda.
Śledzenie rejestratora przebiegu przetwarzania	Informacje na temat śledzenia komponentu Licencjonowanego Kodu Wewnętrznego gniazd będą teraz obejmowały zrzut pozycji rejestratora przebiegu przetwarzania dla każdej operacji przeprowadzonej w gnieździe.
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.
Status połączenia (opcja 3 komendy NETSTAT) udostępniający parametr SO_DEBUG	Udostępnia rozszerzone informacje na temat debugowania niskiego poziomu, jeśli dla aplikacji używającej gniazd zostanie ustawiona opcja SO_DEBUG.
Kod powrotu i przetwarzanie komunikatów gniazd chronionych	Wyświetla standardowe komunikaty kodów powrotu gniazd chronionych za pomocą dwóch funkcji API SSL_. Funkcje te to SSL_Strerror() i SSL_Perror(). Ponadto funkcja API gsk_strerror() udostępnia podobne możliwości co funkcje API GSKit. Dostępna jest także funkcja API hstrerror(), dostarczająca informacji o kodach powrotu procedur programu tłumaczącego.
Śledzenie danych dotyczących wydajności	Udostępnia informacje na temat śledzenia przepływu danych z aplikacji poprzez gniazda i stos TCP/IP.

Informacje pokrewne

Funkcja API SSL_Strerror() - odczytywanie komunikatu o błędzie wykonania w środowisku SSL

Funkcja API SSL_Perror() - drukowanie komunikatu o błędzie w środowisku SSL

Funkcja API gsk_strerror() - odczytywanie komunikatu o błędzie wykonania GSKit

Funkcja API hstrerror() - odczytywanie komunikatu o błędzie programu tłumaczącego

Komenda Uruchomienie śledzenia (Start Trace - STRTRC)

Licencja na kod oraz Informacje dotyczące kodu

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.

Z ZASTRZEŻENIEM GWARANCJI WYNIKAJĄCYCH Z BEZWZGLĘDNIE OBOWIĄZUJĄCYCH PRZEPISÓW PRAWA, IBM, PROGRAMIŚCI ANI DOSTAWCY IBM NIE UDZIELAJĄ NA NINIEJSZY PROGRAM ANI W ZAKRESIE EWENTUALNEGO WSPARCIA TECHNICZNEGO ŻADNYCH GWARANCJI, W TYM TAKŻE RĘKOJMI, NIE USTALAJĄ ŻADNYCH WARUNKÓW, WYRAŹNYCH CZY DOMNIEMANYCH, A W SZCZEGÓLNOŚCI DOMNIEMANYCH GWARANCJI CZY WARUNKÓW PRZYDATNOŚCI HANDLOWEJ, PRZYDATNOŚCI DO OKREŚLONEGO CZY NIENARUSZANIA PRAW STRON TRZECICH.

W ŻADNYCH OKOLICZNOŚCIACH IBM, ANI TEŻ PROGRAMIŚCI CZY DOSTAWCY PROGRAMÓW IBM, NIE PONOSZĄ ODPOWIEDZIALNOŚCI ZA PONIŻSZE SZKODY, NAWET JEŚLI ZOSTALI POINFORMOWANI O MOŻLIWOŚCI ICH WYSTĄPIENIA:

1. UTRATA LUB USZKODZENIE DANYCH;
2. SZKODY BEZPOŚREDNIE, SZCZEGÓLNE, UBOCZNE, POŚREDNIE ORAZ SZKODY, KTÓRYCH NIE MOŻNA BYŁO PRZEWIDZIEĆ PRZY ZAWIERANIU UMOWY, ANI TEŻ
3. UTRATA ZYSKÓW, KONTAKTÓW HANDLOWYCH, PRZYCHODÓW, REPUTACJI (GOODWILL) LUB PRZEWIDYWANYCH OSZCZĘDNOŚCI.

USTAWODAWSTWA NIEKTÓRYCH KRAJÓW NIE DOPUSZCZAJĄ WYŁĄCZENIA CZY OGRANICZENIA ODPOWIEDZIALNOŚCI ZA SZKODY BEZPOŚREDNIE, UBOCZNE LUB SZKODY, KTÓRYCH NIE MOŻNA BYŁO PRZEWIDZIEĆ PRZY ZAWIERANIU UMOWY, W ZWIĄZKU Z CZYM W ODNIESIENIU DO NIEKTÓRYCH KLIENTÓW POWYŻSZE WYŁĄCZENIE LUB OGRANICZENIE (TAK W CAŁOŚCI JAK I W CZĘŚCI) MOŻE NIE MIEĆ ZASTOSOWANIA.

Dodatek. 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
North Castle Drive
Armonk, NY 10504-1785
USA

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
Tokio 106-0032, Japonia

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 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 STRON 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 uwzględnione 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 korzystania i rozpowszechniania informacji przysłanych przez użytkownika w dowolny sposób, jaki uzna za właściwy, bez żadnych zobowiązań wobec ich autora.

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 YBWA
3605 Highway 52 N
Rochester, MN 55901
USA

Informacje takie mogą być udostępnione, o ile spełnione zostaną odpowiednie warunki, w tym, w niektórych przypadkach, uiszczenie odpowiedniej opłaty.

Licencjonowany program opisany w niniejszym dokumencie 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, Umowie Licencyjnej IBM na Kod Maszynowy 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 możliwości 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/nazwiska są fikcyjne i jakiekolwiek podobieństwo do istniejących nazw/nazwisk i adresów jest całkowicie przypadkowe.

LICENCJA W ZAKRESIE PRAW AUTORSKICH:

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 ani sugerować niezawodności, użyteczności i funkcjonalności tych programów.

Każda kopia programu przykładowego lub jakikolwiek jego fragment, jak też jakiekolwiek prace pochodne muszą zawierać następujące uwagi dotyczące praw autorskich:

© (nazwa przedsiębiorstwa użytkownika, rok). Fragmenty tego kodu pochodzą z programów przykładowych IBM Corp. © Copyright IBM Corp. (wpisać rok lub lata). Wszelkie prawa zastrzeżone.

W przypadku przeglądania niniejszych informacji w formie elektronicznej, zdjęcia i kolorowe ilustracje mogą nie być wyświetlane.

Informacje dotyczące interfejsu programistycznego

Niniejsza publikacja opisuje planowane interfejsy programistyczne, pozwalające na pisanie programów umożliwiających korzystanie z usług systemu operacyjnego IBM i5/OS.

Znaki towarowe

Następujące nazwy są znakami towarowymi International Business Machines Corporation w Stanach Zjednoczonych i/lub w innych krajach:

AnyNet
eServer
i5/OS
IBM
IBM (logo)
Integrated Language Environment
iSeries
OS/400
Redbooks
System i

Adobe, logo Adobe, PostScript oraz logo PostScript są zastrzeżonymi znakami towarowymi lub znakami towarowymi firmy Adobe Systems Incorporated 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 Open Group w Stanach Zjednoczonych i w innych krajach.

Nazwy innych przedsiębiorstw, produktów i usług mogą być znakami towarowymi lub znakami usług innych podmiotów.

Warunki

Zezwolenie na korzystanie z tych publikacji jest przyznawane na poniższych warunkach.

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 na ich podstawie 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 na podstawie tych publikacji ani ich fragmentów 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 do anulowania zezwolenia przyznanego w niniejszym dokumencie w każdej sytuacji, gdy, według uznania IBM, korzystanie z tych publikacji jest szkodliwe dla IBM lub jeśli IBM uzna, że warunki niniejszego dokumentu 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 JAKICHKOLWIEK GWARANCJI, W TYM TAKŻE RĘKOJMI, DOTYCZĄCYCH TREŚCI 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, PRZYDATNOŚCI DO OKREŚLONEGO CELU ORAZ NIENARUSZANIA PRAW STRON TRZECICH.

Drukowane w USA