



Systemy IBM - iSeries

Programowanie

Programowanie z użyciem gniazd

Wersja 5 Wydanie 4





Systemy IBM - iSeries

Programowanie

Programowanie z użyciem gniazd

Wersja 5 Wydanie 4

Uwaga

Przed korzystaniem z tych informacji oraz produktu, którego dotyczą, należy przeczytać informacje znajdujące się w sekcji “Uwagi”, na stronie 195.

Wydanie siódme (luty 2006)

Niniejsze wydanie dotyczy Wersji 5, Wydania 3, Modyfikacji 0 systemu IBM i5/OS (5722-SS1) oraz wszelkich kolejnych wersji i modyfikacji tego produktu, o ile nowe wydania nie wskazują inaczej. Wersja ta nie działa na wszystkich modelach komputerów z procesorem RISC ani na modelach z procesorem CISC.

© Copyright International Business Machines Corporation 2001, 2006. Wszelkie prawa zastrzeżone.

Spis treści

Programowanie z użyciem gniazd 1

Co nowego w V5R4	1
Drukowanie dokumentów PDF	1
Wymagania wstępne dla programowania z użyciem gniazd	2
Jak działają gniazda.	3
Charakterystyka gniazd	6
Struktura adresu gniazda	7
Rodzina adresów gniazd	8
Typ gniazda	12
Protokoły gniazd	12
Podstawy projektowania gniazd.	13
Tworzenie gniazd zorientowanych na połączenie.	13
Tworzenie gniazd bezpołączeniowych	21
Projektowanie aplikacji używających rodzin adresów	27
Zaawansowane zagadnienia dotyczące gniazd	43
Asynchroniczne operacje we/wy	43
Gniazda chronione.	46
Obsługa klienta SOCKS	53
Ochrona wątków	57
Nieblokujące operacje we/wy	57
Sygnały	59
Rozsyłanie grupowe IP	60
Przesyłanie danych pliku - send_file() i accept_and_recv()	61
Dane spoza pasma.	62
Multipleksowanie operacji we/wy - funkcja select()	63
Funkcje sieciowe gniazd	63
System nazw domen	64
Zgodność z Berkeley Software Distribution (BSD)	66
Zgodność z UNIX 98..	69
Przekazywanie deskryptorów pomiędzy procesami - sendmsg() i recvmsg()	72
Scenariusze dla gniazd: tworzenie aplikacji komunikujących się z klientami IPv4 i IPv6	74
Przykład: akceptowanie połączeń od klientów IPv6 i IPv4	75

Przykład: klient IPv4 lub IPv6	80
Zalecenia dotyczące projektowania aplikacji używających gniazd	83
Przykłady: projekty aplikacji używających gniazd	86
Przykłady: projekty aplikacji zorientowanych na połączenie	86
Przykład: korzystanie z asynchronicznych operacji we/wy	111
Przykład: nawiązywanie chronionych połączeń	118
Przykład: procedury sieciowe obsługujące ochronę wątków i używające funkcji gethostbyaddr_r()	148
Przykład: nieblokujące operacje we/wy i funkcja select()	151
Używanie funkcji poll() zamiast funkcji select()	157
Przykład: używanie sygnałów z blokującymi funkcjami API gniazd	163
Przykłady: użycie rozsyłania grupowego z obsługą AF_INET	166
Przykład: odpytywanie i aktualizacja serwera DNS	171
Przykład: przesyłanie danych za pomocą funkcji send_file() i accept_and_recv()	176
Narzędzie Xsockets	183
Konfigurowanie Xsockets	183
Konfigurowanie Xsockets do korzystania z przeglądarki WWW	186
Używanie Xsockets	189
Usuwanie obiektów utworzonych przez narzędzie Xsocket	191
Dostosowywanie Xsockets	191
Narzędzia do serwisowania.	191

Dodatek. Uwagi 195

Informacje dotyczące interfejsu programistycznego	197
Znaki towarowe	197
Warunki	197

Programowanie z użyciem gniazd

Gniazdo to miejsce połączenia transmisji (punkt końcowy), które można nazwać i zaadresować w sieci.

Procesy korzystające z gniazd mogą współistnieć w tym samym systemie, jak i w różnych systemach w różnych sieciach. Gniazda są przydatne zarówno dla aplikacji samodzielnych, jak i sieciowych. Gniazda umożliwiają wymianę informacji pomiędzy procesami na tej samej maszynie lub poprzez sieć, dystrybucję zadań do najbardziej wydajnej maszyny oraz ułatwiają dostęp do danych przechowywanych na serwerze. Funkcje API (application program interface) gniazd są standardem dla sieci TCP/IP. Funkcje te są obsługiwane w wielu systemach operacyjnych. Gniazda i5/OS obsługują protokoły sieciowe oraz . Funkcje systemowe i funkcje sieciowe gniazd realizują ochronę wątków.

Programowanie z użyciem gniazd przedstawia wykorzystanie funkcji API gniazd do ustanowienia połączenia między procesem zdalnym i lokalnym. Programiści pracujący w środowisku Integrated Language Environment (ILE) C, mogą korzystać z niniejszych informacji przy tworzeniu aplikacji używających gniazd. Programowanie z wykorzystaniem funkcji API gniazd jest także możliwe dla innych języków w środowisku ILE, na przykład dla RPG.

Język Java również obsługuje interfejs programowania z użyciem gniazd.

Uwaga: Korzystając z przykładowego kodu, użytkownik akceptuje warunki określone w temacie “Informacje dotyczące licencji na kod” na stronie 192.



Co nowego w V5R4

Niniejszy temat dotyczy niektórych ze zmian w programowaniu z użyciem gniazd w systemie V5R4.

- | Dodano obsługę funkcji API poll(). Funkcja API poll() może być używana do określania, kiedy połączenie zostaje zakończone.
- | Dodano zbiorową obsługę rodziny adresów. Nowe opcje gniazda do zbiorowej obsługi IPv6 są dostępne. Więcej szczegółów na ten wyżej wymienionych opcji w temacie setsockopt().
- | IPv6 jest obsługiwana przez wszystkie adaptory Ethernet. Wcześniej, obsługa IPv6 była ograniczona dla adapterów Ethernet 2838. Obecnie, IPv4, IPv6, i PPPoE mogą być używane na jednym adapterze.
- | Nowe urządzenie szyfrujące, 4764 Cryptographic Coprocessor, jest używane do współpracy z SSL.

Gdzie szukać informacji o wprowadzonych zmianach i nowościach

Aby ułatwić użytkownikowi zlokalizowanie wprowadzonych zmian technicznych, dokument ten został oznaczony w następujący sposób:

- Obrazek  wskazuje, gdzie zaczynają się nowości lub zmiany.
- Obrazek  wskazuje, gdzie kończą się nowości lub zmiany.

Dodatkowych informacji na temat wprowadzonych w tym wydaniu systemu zmian i nowości należy szukać w sekcji Informacje dla użytkowników.




Drukowanie dokumentów PDF

Opcja ta służy do oglądania i drukowania dokumentów PDF.











Aby oglądać lub pobrać dokument w formacie PDF, wybierz Programowanie z użyciem gniazd (około 1870 KB).

Inne informacje

Dokumenty te można wyświetlać na ekranie i drukować.

- Java
- Dokumentacja techniczna IBM (Redbooks):
 - Who Knew You can Do That with RPG IV? A Sorcerer's Guide to System Access and More 
 - IBM  iSeries Wired Network Security: OS/400 V5R1 DCM and Cryptographic Enhancements 

Dostępne do przeglądania lub pobrania są następujące pokrewne tematy:


- **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 przeglądania lub drukowania:

1. Najedź na plik PDF w swojej przeglądarce i kliknij prawy przycisk myszy (kliknij powyższy odsyłacz).
2. Kliknij opcję i zapisz plik PDF lokalnie.
3. Przejdź do katalogu, w którym chcesz zapisać plik PDF.
4. Kliknij **Zapisz**.

Pobieranie programu Adobe Reader

Aby przeglądać pliki PDF wymagany jest program Adobe Reader. Darmową wersję programu można pobrać ze strony internetowej Adobe (www.adobe.com/products/acrobat/readstep.html) .

Wymagania wstępne dla programowania z użyciem gniazd

Przed przystąpieniem do pisania aplikacji używającej gniazd należy wykonać następujące zadania:

Wymagania dotyczące kompilatora

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

Wymagania dotyczące rodzin adresów AF_INET i AF_INET6

Oprócz spełnienia wymagań dotyczących kompilatora należy wykonać następujące czynności:

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

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

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

1. Zainstaluj i skonfiguruj program licencjonowany Digital Certificate Manager (5722-SS1 opcja 34). Szczegóły znajdują się w sekcji Digital Certificate Manager w Centrum informacyjnym.
2. Aby używać warstwy SSL ze sprzętem szyfrującym, zainstaluj i skonfiguruj produkt 2058 Cryptographic Accelerator albo 4758 Cryptographic Coprocessor lub 4764 Cryptographic Coprocessor. Produkt 2058 Cryptographic Accelerator umożliwia przeniesienie obciążenia związanego z szyfrowaniem warstwy SSL z systemu operacyjnego na kartę. Produkt 4758 Cryptographic Coprocessor może służyć do szyfrowania warstwy SSL, jednakże w przeciwieństwie do produktu 2058 udostępnia więcej funkcji szyfrujących, na przykład szyfrowanie i deszyfrowanie kluczy. Produkt 4764 Cryptographic Coprocessor jest lepszą wersją produktu 4758 Cryptographic Coprocessor. Sekcja Sprzęt szyfrujący zawiera kompletne opisy produktów 2058 Cryptographic Accelerator, 4758 Cryptographic Coprocessor, i produktu 4764 Cryptographic Coprocessor.

Odsyłacze pokrewne

“Używanie rodziny adresów AF_INET” na stronie 27

Gniazda rodziny adresów AF_INET mogą być zorientowane na połączenie (typu SOCK_STREAM), jak i bezpołączeniowe (typu SOCK_DGRAM).

“Używanie rodziny adresów AF_INET6” na stronie 27

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

“Używanie rodziny adresów AF_UNIX” na stronie 29

Gniazda rodziny adresów AF_UNIX (gniazda używające rodzin adresów AF_UNIX lub AF_UNIX_CCSDID) mogą być zorientowane na połączenie (typu SOCK_STREAM) lub bezpołączeniowe (typu SOCK_DGRAM).

Jak działają gniazda

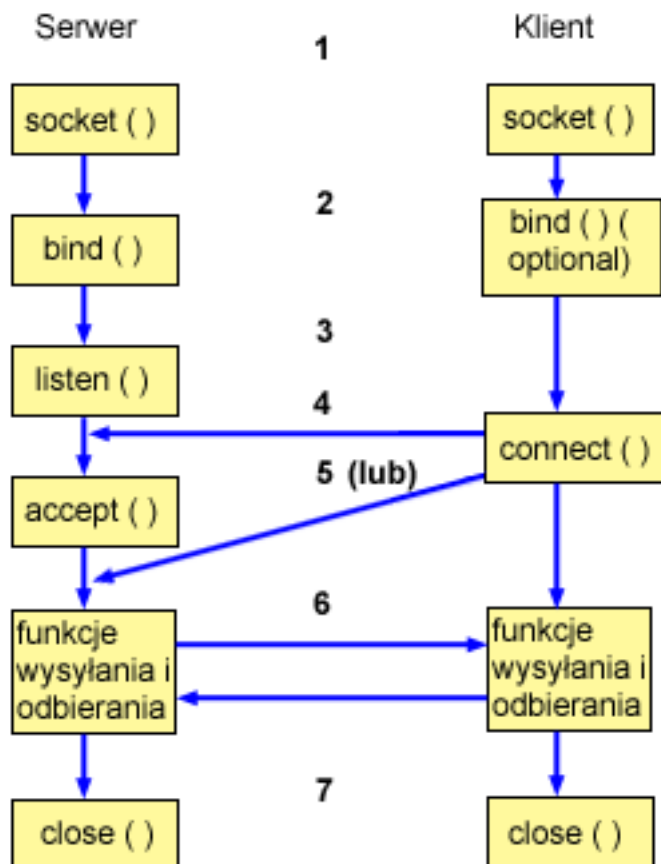
Gniazda są powszechnie używane w interakcjach klient/serwer. Typowa konfiguracja systemu umieszcza serwer na jednym komputerze, a klientów na innych komputerach. Klienci łączą się z serwerem, wymieniają informacje, a następnie się odłączają.

Ustanawianie gniazd wiąże się z typowym przebiegiem zdarzeń. W zorientowanym na połączenie modelu klient-serwer gniazdo procesu serwera czeka na żądania od klienta. W tym celu serwer najpierw ustanawia (wiąże) adres, z którego klient może skorzystać, aby znaleźć serwer. Gdy adres jest ustanowiony, wtedy serwer czeka, aż klient zażąda usługi.

Wymiana danych od klienta do serwera zachodzi wtedy, gdy klient łączy się z serwerem poprzez gniazdo. Serwer spełnia żądanie klienta i wysyła odpowiedź z powrotem do klienta.

Uwaga: Obecnie, IBM obsługuje dwie wersje większości funkcji API gniazd. Domyślna funkcja API systemu i5/OS używa struktury i składni BSD (Berkeley Socket Distribution) 4.3. Druga wersja gniazd używa składni i struktur zgodnych z 4.4 i specyfikacją interfejsu programistycznego UNIX 98. Aby korzystać z interfejsu zgodnego ze standardem UNIX 98, programiści mogą określić makrodefinicję XOPEN SOURCE.

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

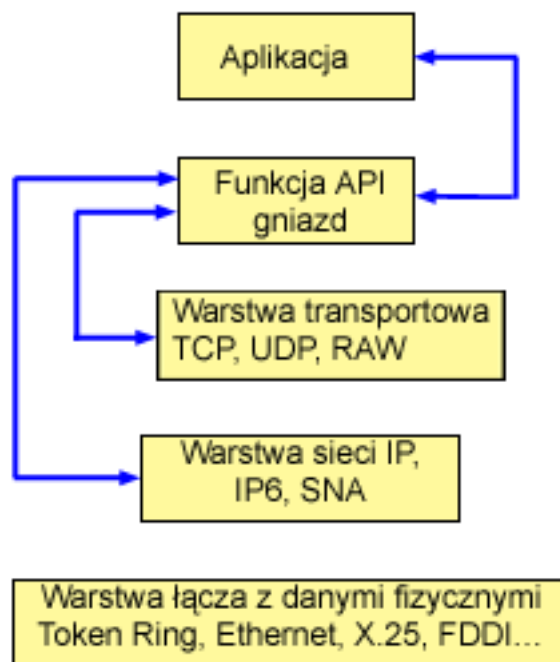


Typowy przebieg zdarzeń dla gniazd zorientowanych na połączenie:

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

7. Gdy jedna ze stron chce zakończyć działanie, wywołuje funkcję `close()`, która zwalnia wszystkie przydzielone gniazdu zasoby systemu.

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



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

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

Odsyłacze pokrewne

“Zgodność z Berkeley Software Distribution (BSD)” na stronie 66

Gniazda są interfejsem systemu (BSD).

“Zgodność z UNIX 98.” na stronie 69

Opracowany przez Open Group, konsorcjum programistów i dostawców oprogramowania, standard UNIX 98 usprawnił współdziałanie systemu operacyjnego UNIX zawierając przy tym większość funkcji związanych z obsługą Internetu, dzięki którym system UNIX stał się znany.

Informacje pokrewne

`socket()`

`listen()`

`bind()`

`accept()`

`send()`

`recv()`

`close()`

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 odniesienie do niego.
- Gniazdo można nazwać i wykorzystywać do komunikacji z innymi gniazdami w domenie komunikacyjnej.
- Gniazda komunikują się, gdy serwer przyjmuje od nich połączenia lub gdy wymieniają komunikaty z serwerem.
- Gniazda można tworzyć parami (tylko dla gniazd w rodzinie adresów AF_UNIX).

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

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

Określanie charakterystyk gniazd

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

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

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

Tabela 1. Podsumowanie charakterystyk gniazd

Rodzina adresów	Typ gniazda	Protokół gniazda
AF_UNIX	SOCK_STREAM	nie dotyczy
	SOCK_DGRAM	nie dotyczy
AF_INET	SOCK_STREAM	TCP
	SOCK_DGRAM	UDP
	SOCK_RAW	IP, ICMP
AF_INET6	SOCK_STREAM	TCP
	SOCK_DGRAM	UDP

Tabela 1. Podsumowanie charakterystyk gniazd (kontynuacja)

Rodzina adresów	Typ gniazda	Protokół gniazda
	SOCK_RAW	IP6, ICMP6
AF_UNIX_CCSID	SOCK_STREAM	nie dotyczy
	SOCK_DGRAM	nie dotyczy

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

Funkcje API gniazd

Struktura adresu gniazda

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

Obecnie, system i5/OS obsługuje supports gniazda (BSD) 4.3 i X/Open Single Unix Specification (UNIX 98). Podstawowa funkcja API systemu i5/OS używa struktury i składni BSD 4.3. Definiując wartość 520 lub większą dla makra `_XOPEN_SOURCE`, można wybrać interfejs zgodny ze standardem UNIX 98. Dla każdej użytej struktury gniazd w standardzie BSD 4.3 istnieje odpowiednik w strukturze UNIX 98.

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

Struktura BSD 4.3	Struktura BSD 4.4 kompatybilna z systemem UNIX 98
<pre>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]; };</pre>	<pre>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]; };</pre>

Tabela 3. Struktura adresów

Pole struktury adresu	Definicja
sa_len	Pole to zawiera długość adresu wg specyfikacji UNIX 98. Uwaga: Pole sa_len służy wyłącznie do zapewnienia zgodności ze standardem BSD 4.4. Nie ma potrzeby używania go, nawet w przypadku korzystania ze zgodności ze standardami BSD 4.4/UNIX 98. Pole jest ignorowane w przypadku adresów wejściowych.

Tabela 3. Struktura adresów (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 jest określony typem transportu, dla którego zostało utworzone gniazdo. Każdy z protokołów warstwy transportowej definiuje dokładny format, odpowiadający jego wymaganiom, w podobnej strukturze adresu. Protokół transportowy jest identyfikowany przez wartość parametru protokołu dla funkcji API socket().
sockaddr_storage	Deklaruje obszar pamięci dla dowolnego adresu z tej rodziny adresów. Struktura ta jest wystarczająco duża i dopasowana do wszystkich struktur zależnych od użytego protokołu. Można zatem podczas korzystania z funkcji API rzutować ją jako strukturę as_sockaddr. Pole ss_family struktury sockaddr_storage zawsze będzie dopasowane do pola rodziny wszystkich struktur zależnych od protokołu.

Rodzina adresów gniazd

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

Protokoły rodziny adresów zapewniają transport sieciowy danych aplikacji z jednej aplikacji do innej (lub z jednego procesu do innego wewnątrz tej samej maszyny). Aplikacja określa protokół transportu sieciowego w parametrze protokołu gniazda.

Parametr rodziny adresów (address_family) dla funkcji socket() określa strukturę adresu używaną przez funkcje gniazd.

Rodzina adresów AF_INET

Rodzina adresów AF_INET umożliwia komunikację międzyprocesową pomiędzy procesami działającymi w tym samym systemie lub w różnych systemach.

Adresy dla gniazd AF_INET to adresy IP i numery portów. Adres dla gniazda AF_INET można podać w postaci adresu IP, na przykład 130.99.128.1, lub w formie 32-bitowej, czyli X'82638001'.

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

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

Struktura adresów sockaddr wg specyfikacji BSD 4.3	Struktura adresów sockadrr wg specyfikacji BSD 4.4/ UNIX 98
<pre>struct sockaddr_in { short sin_family; u_short sin_port; struct in_addr sin_addr; char sin_zero[8]; };</pre>	<pre>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]; };</pre>

Tabela 5. Struktura adresów AF_INET

Pole struktury adresu	Definicja
sin_len	Pole to zawiera długość adresu wg specyfikacji UNIX 98. Uwaga: Pole sin_len służy wyłącznie do zapewnienia zgodności ze standardem BSD 4.4. Nie ma potrzeby używania go, nawet w przypadku korzystania ze zgodności ze standardami BSD 4.4/ UNIX. 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 pole należy wpisać szesnastkowe zera.

Odsyłacze pokrewne

“Używanie rodziny adresów AF_INET” na stronie 27

Gniazda rodziny adresów AF_INET mogą być zorientowane na połączenie (typu SOCK_STREAM), jak i bezpołączeniowe (typu SOCK_DGRAM).

Rodzina adresów AF_INET6

Ta rodzina adresów zapewnia obsługę protokołu IP (Internet Protocol) w wersji 6 (IPv6). Rodzina adresów AF_INET6 używa adresów 128-bitowych (16-bajtowych).

W uproszczeniu, architektura tych adresów obejmuje 64 bity numeru sieci i 64 bity numeru hosta. Adresy z rodziny AF_INET6 można podawać w postaci x:x:x:x:x:x:x, gdzie 'x' oznacza szesnastkowe wartości ośmiu 16-bitowych części adresu. Przykładowo, adres może wyglądać następująco: FEDC:BA98:7654:3210:FEDC:BA98:7654:3210.

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

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

Struktura adresów sockaddr_in6 wg specyfikacji BSD 4.3	Struktura adresów sockadrr in6 wg specyfikacji BSD 4.4/ UNIX 98
<pre>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; };</pre>	<pre>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; };</pre>

Tabela 7. Struktura adresów AF_INET6

Pole struktury adresu	Definicja
sin6_len	Pole to zawiera długość adresu wg specyfikacji UNIX 98. Uwaga: Pole sin6_len służy wyłącznie do zapewnienia zgodności ze standardem BSD 4.4. Nie ma potrzeby używania go, nawet w przypadku korzystania ze zgodności ze standardami BSD 4.4/ UNIX. Pole jest ignorowane w przypadku adresów wejściowych.
sin6_family	Pole to określa, że zostanie użyta rodzina adresów AF_INET6.

Tabela 7. Struktura adresów AF_INET6 (kontynuacja)

Pole struktury adresu	Definicja
sin6_port	Pole to zawiera port warstwy protokołu transportowego.
sin6_flowinfo	Pole to zawiera dwie informacje: klasę ruchu i etykietę przepływu. Uwaga: Nie jest ono obecnie obsługiwane i aby zapewnić zgodność z przyszłymi systemami, należy ustawić jego wartość na zero.
sin6_addr	Pole to określa adres IPv6.
sin6_scope_id	Pole to identyfikuje zestaw interfejsów odpowiednich dla zakresu adresów określonych w polu sin6_addr .

Rodzina adresów AF_UNIX

Rodzina adresów AF_UNIX umożliwia komunikację międzyprocesową w ramach jednego systemu, w którym używane są funkcje API gniazd. Adres jest w rzeczywistości nazwą ścieżki do pozycji systemu plików.

Gniazda można tworzyć w katalogu głównym lub w dowolnym otwartym systemie plików, na przykład asQSYS lub QDOC. Aby odbierać odsyłane datagramy, program musi powiązać gniazdo AF_UNIX, SOCK_DGRAM z nazwą. Dodatkowo, po zamknięciu gniazda, program musi w sposób jawny usunąć obiekt systemu plików funkcją API unlink().

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

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

Struktura adresów sockadr_un wg specyfikacji BSD 4.3	Struktura adresów sockadr_un wg specyfikacji BSD 4.4/ UNIX 98
<pre>struct sockadr_un { short sun_family; char sun_path[126]; };</pre>	<pre>struct sockadr_un { uint8_t sun_len; sa_family_t sun_family; char sun_path[126]; };</pre>

Tabela 9. Struktura adresu AF_UNIX

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

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

Odsyłacze pokrewne

“Używanie rodziny adresów AF_UNIX” na stronie 29

Gniazda rodziny adresów AF_UNIX (gniazda używające rodzin adresów AF_UNIX lub AF_UNIX_CC SID) mogą być zorientowane na połączenie (typu SOCK_STREAM) lub bezpołączeniowe (typu 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

unlink()

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 i żadne zewnętrzne funkcje komunikacyjne nie są używane do łączności między procesami. Różnica między rodzinami polega na tym, że rodzina adresów AF_UNIX_CCSID korzysta ze struktury adresu sockaddr_unc. Ta struktura adresu jest podobna do sockaddr_un, ale umożliwia używanie nazwy ścieżki w kodzie UNICODE lub w dowolnym identyfikatorze CCSID, poprzez użycie formatu Qlg_Path_Name_T.

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

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

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

Tabela 10. Struktura adresu AF_UNIX_CCSID

Pole struktury adresu	Definicja
sunc_family	Rodzina adresów; w tym przypadku zawsze AF_UNIX_CCSID.
sunc_format	To pole zawiera dwie określone wartości dla formatu nazwy ścieżki: <ul style="list-style-type: none">SO_UNC_DEFAULT oznacza długą nazwę ścieżki z użyciem bieżącego identyfikatora CCSID dla nazw ścieżek zintegrowanego systemu plików. Pole sunc_qlg jest ignorowane.SO_UNC_USE_QLG oznacza, że pole sunc_qlg definiuje format i identyfikator CCSID nazwy ścieżki.
sunc_zero	Pole zastrzeżone. W pole należy wpisać szesnastkowe zera.
sunc_qlg	Format nazwy ścieżki.
sunc_path	Nazwa ścieżki. Maksymalna długość nazwy wynosi 126 znaków i może być jedno- lub dwubajtowa. Nazwa może być zawarta w polu sunc_path field lub przydzielona osobno i wskazana poprzez wartość w polu sunc_path. Format nazwy jest określony wartościami pól sunc_format i sunc_qlg.

Odsyłacze pokrewne

“Używanie rodziny adresów AF_UNIX_CCSID” na stronie 36

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

“Rodzina adresów AF_UNIX” na stronie 10

Rodzina adresów AF_UNIX umożliwia komunikację międzyprocesową w ramach jednego systemu, w którym używane są funkcje API gniazd. Adres jest w rzeczywistości nazwą ścieżki do pozycji systemu plików.

Informacje pokrewne

Format nazwy ścieżki

Typ gniazda

Drugi parametr w wywołaniu gniazda określa typ gniazda. Typ gniazda umożliwia identyfikację typu i charakterystyki połączenia, które zostanie nawiązane w celu transportu danych z jednej maszyny do innej lub z jednego procesu do innego.

Poniższa lista opisuje typy gniazd obsługiwane przez serwer iSeries:

Strumieniowe (SOCK_STREAM)

Ten rodzaj gniazda jest zorientowany na połączenie. Nawiązanie połączenia na całej trasie następuje przy użyciu funkcji `bind()`, `listen()`, `accept()`, i `connect()`. SOCK_STREAM wysyła dane bez błędów czy powtórzeń i otrzymuje dane w kolejności wysyłania. SOCK_STREAM stosuje sterowanie przepływem, aby uniknąć przekroczenia granicy danych. Nie narzuca granic bloków dla danych. Zakłada, że dane stanowią strumień bajtów. W implementacji iSeries gniazda strumieniowych można używać w sieciach Transmission Control Protocol (TCP), AF_UNIX i AF_UNIX_CCSID. Gniazda strumieniowych można także używać do połączeń się z systemami poza hostem chronionym (firewallem).

Datagram (SOCK_DGRAM)

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

Surowe (SOCK_RAW)

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

Protokoły gniazd

Protokoły umożliwiają sieciowy transport danych aplikacji z jednej maszyny do innej (lub z jednego procesu do innego).

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

Używając rodziny adresów można korzystać z kilku protokołów transportowych. Protokoły architektury systemów sieciowych (SNA) i TCP/IP mogą być aktywne używając tego samego gniazda nasłuchującego jednocześnie. Atrybut

sieciowy ALWANYNET (Allow ANYNET support - Zezwolenie na obsługę AnyNet) umożliwia wybranie innego transportu niż TCP/IP dla aplikacji używających gniazd AF_INET. Atrybut ten może mieć wartość *YES lub *NO. Wartością domyślną jest *NO.

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

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

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

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

Informacje pokrewne

Konfiguracja APPC, APPN, i HPR

Podstawy projektowania gniazd

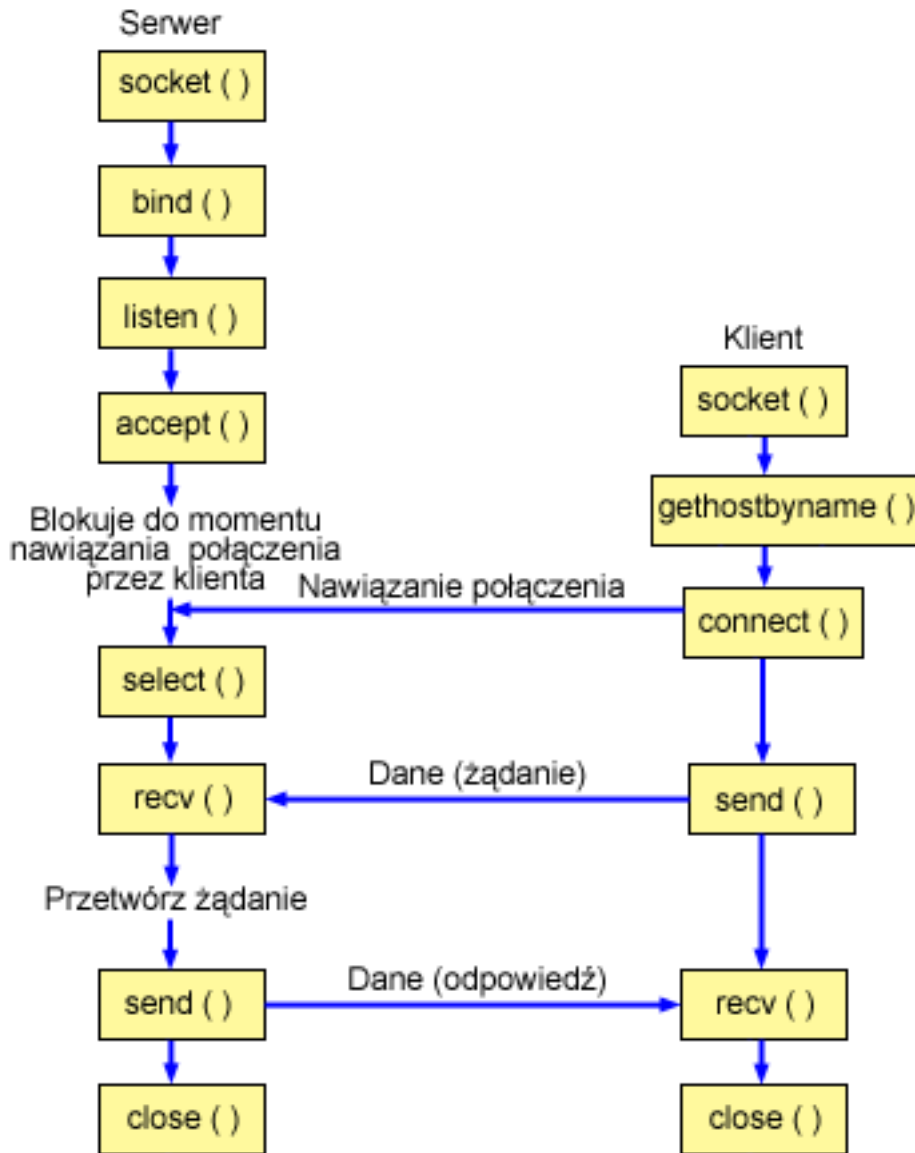
W tej sekcji przedstawiono przykłady programów, w których zaimplementowano najprostsze gniazda.

Przykłady te stanowią podstawę dla konstrukcji bardziej zaawansowanych. Następujące programy stanowią przykłady najpopularniejszych typów programów używających gniazd.

Tworzenie gniazd zorientowanych na połączenie

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

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



Przebieg zdarzeń w gnieździe: serwer zorientowany na połączenie

Na poniższej ilustracji wyjaśniono sekwencję wywołań funkcji gniazd. Opisano także relacje pomiędzy aplikacją serwera a aplikacją klienta w architekturze zorientowanej na połączenie. Każdy zbiór wywołań zawiera odsyłacze do uwag dotyczących użycia poszczególnych funkcji API.

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

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

Przebieg zdarzeń w gnieździe: klient zorientowany na połączenie

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

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

Odsyłacze pokrewne

“Przykład: program serwera zorientowany na połączenie”

Poniższy przykładowy kod pokazuje, jak można utworzyć serwer zorientowany na połączenie.

Informacje pokrewne

`listen()`

`bind()`

`accept()`

`send()`

`recv()`

`close()`

`socket()`

`setsockopt()`

`select()`

`gethostbyname()`

`connect()`

Przykład: program serwera zorientowany na połączenie

Poniższy przykładowy kod pokazuje, jak można utworzyć serwer zorientowany na połączenie.

Na jego podstawie można utworzyć własną aplikację serwera gniazd. Serwer zorientowany na połączenie jest jednym z najczęściej używanych modeli aplikacji używających gniazd. W modelu zorientowanym na połączenie aplikacja serwera tworzy gniazdo, które służy do odbierania żądań z klientów.

Uwaga: Korzystając z przykładowego kodu, użytkownik akceptuje warunki określone w temacie “Informacje dotyczące licencji na kod” na stronie 192.

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

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

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

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

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

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

```

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

/*****
/* Po utworzeniu deskryptora gniazda funkcja bind() pobiera */
/* unikalną nazwę gniazda. W tym przykładzie użytkownik ustawia */
/* wartość s_addr na zero, co umożliwia nawiązanie połączenia z */
/* dowolnego klienta, który określi port 3005. */
*****/
memset(&serveraddr, 0, sizeof(serveraddr));
serveraddr.sin_family = AF_INET;
serveraddr.sin_port = htons(SERVER_PORT);
serveraddr.sin_addr.s_addr = htonl(INADDR_ANY);

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

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

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

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

/*****
/* Funkcja select() powoduje, że proces oczekuje na wystąpienie */
/* zdarzenia, po którym kontynuuje działanie. W tym przykładzie */
/* system powiadamia proces tylko wtedy, gdy dane do odczytania */
/* będą dostępne. W tym wywołaniu funkcji select() został określony */
/* limit czasu równy 30 sek. */
*****/
timeout.tv_sec = 30;
timeout.tv_usec = 0;

FD_ZERO(&read_fd);
FD_SET(sd2, &read_fd);

rc = select(sd2+1, &read_fd, NULL, NULL, &timeout);
if (rc < 0)
{
    perror("Niepowodzenie select()");
}

```

```

    break;
}

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

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

if (rc < 0)
{
    perror("Niepowodzenie setsockopt(SO_RCVLOWAT)");
    break;
}

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

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

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

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

} while (FALSE);

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

```


Odsyłacze pokrewne

“Tworzenie gniazd zorientowanych na połączenie” na stronie 13

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

“Używanie rodziny adresów AF_INET” na stronie 27

Gniazda rodziny adresów AF_INET mogą być zorientowane na połączenie (typu SOCK_STREAM), jak i bezpołączeniowe (typu SOCK_DGRAM).

Przykład: program klienta zorientowany na połączenie

Poniższy przykład ilustruje tworzenie programu klienta gniazd do łączenia z serwerem zorientowanym na połączenie w systemie zorientowanym na połączenie.

Klient usługi (program typu klient) musi zażądać usługi od programu typu serwer. W oparciu o ten przykładowy kod można pisać własne aplikacje typu klient.

Uwaga: Korzystając z przykładowego kodu, użytkownik akceptuje warunki określone w temacie “Informacje dotyczące licencji na kod” na stronie 192.

```
/******  
/* Jest to przykładowy kod programu klienta zorientowanego na połączenie. */  
/******  
  
/******  
/* Wymagane przez program pliki nagłówkowe. */  
/******  
#include <stdio.h>  
#include <string.h>  
#include <sys/types.h>  
#include <sys/socket.h>  
#include <netinet/in.h>  
#include <arpa/inet.h>  
#include <netdb.h>  
  
/******  
/* Stałe używane przez ten program */  
/******  
#define SERVER_PORT 3005  
#define BUFFER_LENGTH 250  
#define FALSE 0  
#define SERVER_NAME "NazwaSerweraHosta"  
  
/* Przekaż 1 parametr, który albo jest */  
/* adresem, albo nazwą hosta serwera lub */  
/* ustaw nazwę serwera zmienną #define */  
/* SERVER_NAME. */  
void main(int argc, char *argv[])  
{  
    /******  
    /* Definicje zmiennych i struktur. */  
    /******  
    int sd=-1, rc, bytesReceived;  
    char buffer[BUFFER_LENGTH];  
    char server[NETDB_MAX_HOST_NAME_LENGTH];  
    struct sockaddr_in serveraddr;  
    struct hostent *hostp;  
  
    /******  
    /* Pętla do/while(FALSE) ułatwia realizację procedur czyszczących w */  
    /* przypadku błędu. Funkcja close() dla deskryptora gniazda jest */  
    /* uruchamiana jednokrotnie na samym końcu programu. */  
    /******  
    do  
    {  
        /******
```

```

/* Funkcja socket() zwraca deskryptor gniazda stanowiącego punkt */
/* końcowy. Instrukcja ta określa także, że dla tego gniazda */
/* użyta zostanie rodzina adresów INET (Internet Protocol) */
/* z protokołem transportowym TCP (SOCK_STREAM). */
/*****
sd = socket(AF_INET, SOCK_STREAM, 0);
if (sd < 0)
{
perror("Niepowodzenie socket()");
break;
}

/*****
/* Jeśli został przekazany argument, należy go użyć jako nazwy */
/* serwera, w przeciwnym razie należy użyć zmiennej określonej */
/* w makrze #define znajdującym się na początku programu. */
/*****
if (argc > 1)
strncpy(server, argv[1]);
else
strncpy(server, SERVER_NAME);

memset(&serveraddr, 0, sizeof(serveraddr));
serveraddr.sin_family = AF_INET;
serveraddr.sin_port = htons(SERVER_PORT);
serveraddr.sin_addr.s_addr = inet_addr(server);
if (serveraddr.sin_addr.s_addr == (unsigned long)INADDR_NONE)
{
/*****
/* łańcuch określający serwer, przekazany do funkcji inet_addr() */
/* nie jest adresem IP składającym się z liczb dziesiętnych */
/* oddzielonych kropkami, zatem musi być to nazwa hosta serwera. */
/* Do pobrania adresu IP serwera zostanie użyta funkcja */
/* gethostbyname(). */
/*****
hostp = gethostbyname(server);
if (hostp == (struct hostent *)NULL)
{
printf("Hosta nie znaleziono --> ");
printf("h_errno = %d\n", h_errno);
break;
}

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

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

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

```

```

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

    /******
    /* W tym przykładzie wiadomo, że serwer odpowie wysłaniem tych */
    /* samych 250 bajtów, które 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 przysyłanych w oddzielnych pakietach, */
    /* funkcja recv() będzie wielokrotnie uruchamiana, dopóki */
    /* wszystkie 250 bajtów nie zostanie odebrane. */
    /******
    bytesReceived = 0;
    while (bytesReceived < BUFFER_LENGTH)
    {
        rc = recv(sd, & buffer[bytesReceived],
                BUFFER_LENGTH - bytesReceived, 0);
        if (rc < 0)
        {
            perror("Niepowodzenie recv()");
            break;
        }
        else if (rc == 0)
        {
            printf("Serwer zamknął połączenie\n");
            break;
        }

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

} while (FALSE);

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

```

Odsyłacze pokrewne

“Używanie rodziny adresów AF_INET” na stronie 27

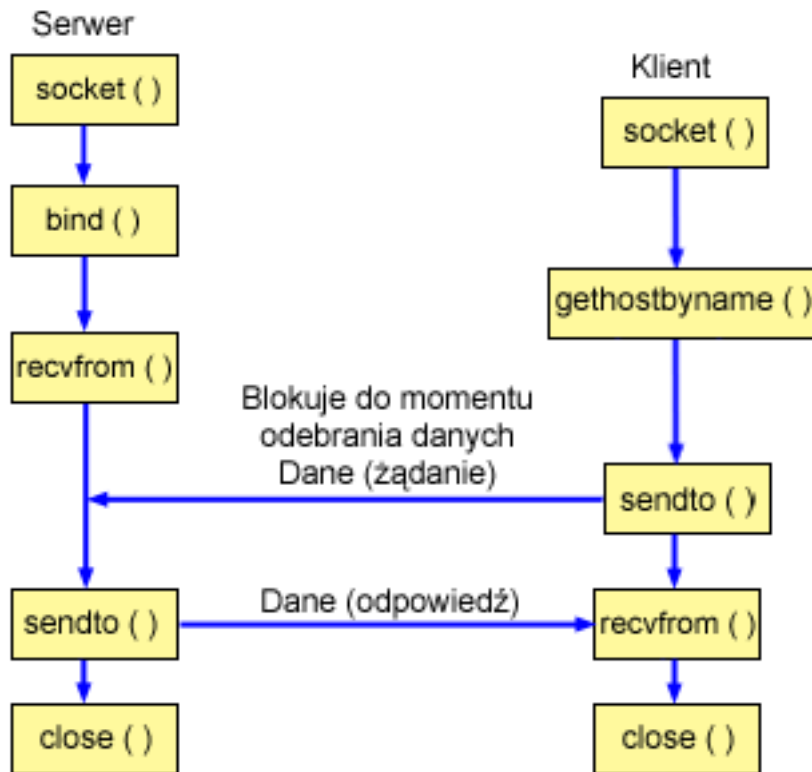
Gniazda rodziny adresów AF_INET mogą być zorientowane na połączenie (typu SOCK_STREAM), jak i bezpołączeniowe (typu SOCK_DGRAM).

Tworzenie gniazd bezpołączeniowych

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

Gniazda bezpołączeniowe zamiast protokołu TCP/IP używają protokołu UDP.

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



Przebieg zdarzeń w gnieździe: serwer bezpołączeniowy

Poniżej wyjaśniono sekwencję wywołań funkcji gniazd, przedstawionych na powyższej ilustracji i w programach przykładowych. Opisano także relacje pomiędzy aplikacją serwera a aplikacją klienta w architekturze bezpołączeniowej. Każdy zbiór wywołań zawiera odsyłacze do uwag dotyczących użycia poszczególnych funkcji API. Aby uzyskać szczegółowe informacje dotyczące użycia tych funkcji API, należy użyć poniższych odsyłaczy. Pierwszy przykład wykorzystuje następującą sekwencję wywołań funkcji:

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

Przebieg zdarzeń w gnieździe: klient bezpołączeniowy

Kolejny przykład wykorzystuje następującą sekwencję wywołań funkcji:

1. Funkcja `socket()` zwraca deskryptor gniazda odpowiadający punktowi końcowemu. Instrukcja ta informuje również, że dla tego gniazda zostanie użyta rodzina adresów INET (Internet Protocol) z transportem UDP transport (`SOCK_DGRAM`).
2. W przykładowym programie klienta, jeśli przesłany do funkcji `inet_addr()` ciąg znaków serwera nie jest adresem IP składającym się z liczb dziesiętnych oddzielonych kropkami, zakłada się, że jest to nazwa hosta serwera. W takim przypadku do pobrania adresu IP serwera używa się funkcji `gethostbyname()`.

3. Funkcja `sendto()` odsyła dane do serwera.
4. Do odebrania danych z powrotem z serwera zostanie użyta funkcja `recvfrom()`.
5. Funkcja `close()` zamyka wszystkie otwarte deskryptory gniazd.

Informacje pokrewne

`close()`
`socket()`
`bind()`
`recvfrom()`
`sendto()`
`gethostbyname()`

Przykład: bezpołączeniowy program serwera

Tego przykładu można użyć do tworzenia własnego serwera bezpołączeniowego. Poniższy program bezpołączeniowego serwera gniazd używa protokołu UDP.

Uwaga: Korzystając z przykładowego kodu, użytkownik akceptuje warunki określone w temacie “Informacje dotyczące licencji na kod” na stronie 192.

```

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

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

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

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

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

```

```

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

    /*****/
    /* Po utworzeniu deskryptora gniazda funkcja bind() pobiera */
    /* unikalną nazwę gniazda. W tym przykładzie użytkownik ustawia */
    /* wartość s_addr na zero, co oznacza, że port UDP 3555 będzie */
    /* powiązany z wszystkimi adresami IP w systemie. */
    /*****/
    memset(&serveraddr, 0, sizeof(serveraddr));
    serveraddr.sin_family = AF_INET;
    serveraddr.sin_port = htons(SERVER_PORT);
    serveraddr.sin_addr.s_addr = htonl(INADDR_ANY);

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

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

    if (rc < 0)
    {
        perror("Niepowodzenie recvfrom()");
        break;
    }

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

    /*****/
    /* Odesłanie danych do klienta */
    /*****/
    rc = sendto(sd, buffer, sizeof(buffer), 0,
                (struct sockaddr *)&clientaddr,
                sizeof(clientaddr));

    if (rc < 0)
    {
        perror("Niepowodzenie sendto()");
        break;
    }

    /*****/
    /* Zakonczenie programu */
    /*****/

} while (FALSE);

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

```

Przykład: bezpołączeniowy program klienta

Poniższy przykład ilustruje użycie protokołu UDP do łączenia bezpołączeniowego programu klienta używającego gniazd z serwerem.

Uwaga: Korzystając z przykładowego kodu, użytkownik akceptuje warunki określone w temacie “Informacje dotyczące licencji na kod” na stronie 192.

```
/* *****  
/* Jest to przykładowy kod programu klienta bezpołączeniowego. */  
/* *****  
  
/* *****  
/* Wymagane przez program pliki nagłówkowe. */  
/* *****  
#include <stdio.h>  
#include <string.h>  
#include <sys/types.h>  
#include <sys/socket.h>  
#include <netinet/in.h>  
#include <arpa/inet.h>  
#include <netdb.h>  
  
/* *****  
/* Stałe używane przez ten program */  
/* *****  
#define SERVER_PORT 3555  
#define BUFFER_LENGTH 100  
#define FALSE 0  
#define SERVER_NAME "NazwaSerweraHosta"  
  
/* Przekaż 1 parametr, który albo jest */  
/* adresem, albo nazwą hosta serwera lub */  
/* ustaw nazwę serwera zmienną #define */  
/* SERVER_NAME. */  
void main(int argc, char *argv[])  
{  
    /* *****  
    /* Definicje zmiennych i struktur. */  
    /* *****  
    int sd, rc;  
    char server[NETDB_MAX_HOST_NAME_LENGTH];  
    char buffer[BUFFER_LENGTH];  
    struct hostent *hostp;  
    struct sockaddr_in serveraddr;  
    int serveraddrlen = sizeof(serveraddr);  
  
    /* *****  
    /* Pętla do/while(FALSE) ułatwia realizację procedur czyszczących w */  
    /* przypadku błędu. Funkcja close() dla deskryptora gniazda jest */  
    /* uruchamiana jednokrotnie na samym końcu programu. */  
    /* *****  
    do  
    {  
        /* *****  
        /* Funkcja socket() zwraca deskryptor gniazda stanowiącego punkt */  
        /* końcowy. Instrukcja ta określa także, że dla tego gniazda */  
        /* użyta zostanie rodzina adresów INET (Internet Protocol) */  
        /* z protokołem transportowym TCP (SOCK_STREAM). */  
        /* *****  
        sd = socket(AF_INET, SOCK_DGRAM, 0);  
        if (sd < 0)  
        {  
            perror("Niepowodzenie socket()");  
            break;  
        }  
    }  
}
```

```

/*****
/* Jeśli został przekazany argument, należy go użyć jako nazwy
/* serwera, w przeciwnym razie należy użyć zmiennej określonej
/* w makrze #define znajdującym się na początku programu.
*****/
if (argc > 1)
    strcpy(server, argv[1]);
else
    strcpy(server, SERVER_NAME);

memset(&serveraddr, 0, sizeof(serveraddr));
serveraddr.sin_family = AF_INET;
serveraddr.sin_port = htons(SERVER_PORT);
serveraddr.sin_addr.s_addr = inet_addr(server);
if (serveraddr.sin_addr.s_addr == (unsigned long)INADDR_NONE)
{
/*****
/* Łańcuch określający serwer, przekazany do funkcji inet_addr()
/* nie jest adresem IP składającym się z liczb dziesiętnych
/* oddzielonych kropkami, zatem musi być to nazwa hosta serwera.
/* Do pobrania adresu IP serwera zostanie użyta funkcja
/* gethostbyname().
*****/
hostp = gethostbyname(server);
if (hostp == (struct hostent *)NULL)
{
    printf("Hosta nie znaleziono --> ");
    printf("h_errno = %d\n", h_errno);
    break;
}

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

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

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

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

```



```

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

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

} while (FALSE);

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

```

Projektowanie aplikacji używających rodzin adresów

Niniejsza sekcja zawiera przykłady programów, które ilustrują każdą rodzinę adresów gniazd.

Używanie rodziny adresów AF_INET

Gniazda rodziny adresów AF_INET mogą być zorientowane na połączenie (typu SOCK_STREAM), jak i bezpołączeniowe (typu SOCK_DGRAM).

Zorientowane na połączenie gniazda AF_INET używają TCP jako protokołu transportowego. Gniazda bezpołączeniowe zamiast protokołu TCP/IP używają protokołu 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

Rodzina adresów AF_INET umożliwia komunikację międzyprocesową pomiędzy procesami działającymi w tym samym systemie lub w różnych systemach.

“Wymagania wstępne dla programowania z użyciem gniazd” na stronie 2

Przed przystąpieniem do pisania aplikacji używającej gniazd należy wykonać następujące zadania:

“Przykład: program serwera zorientowany na połączenie” na stronie 15

Poniższy przykładowy kod pokazuje, jak można utworzyć serwer zorientowany na połączenie.

“Przykład: program klienta zorientowany na połączenie” na stronie 19

Poniższy przykład ilustruje tworzenie programu klienta gniazd do łączenia z serwerem zorientowanym na połączenie w systemie zorientowanym na połączenie.

Używanie rodziny adresów AF_INET6

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

Podobnie jak gniazda AF_INET, gniazda rodziny adresów AF_INET6 mogą być zorientowane na połączenie (typu SOCK_STREAM), jak i bezpołączeniowe (typu SOCK_DGRAM). Zorientowane na połączenie gniazda AF_INET6 jako protokołu transportowego używają TCP. Gniazda bezpołączeniowe AAAAF_INET6 zamiast używają protokołu 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 (Internet Protocol) wersja 6 (IPv6), współpracują z aplikacjami dla protokołu IP (Internet Protocol) wersja 4 (IPv4), czyli używającymi rodziny adresów AF_INET. Dzięki temu programiści piszący aplikacje gniazd mogą korzystać z formatu adresu IPv4 odwzorowanego na adres IPv6. Format ten polega na tym, że adresowi IPv4 węzła IPv4 odpowiada adres IPv6. Adres IPv4 jest zapisywany w najmłodszych 32 bitach adresu IPv6, a najstarsze 96 bitów stanowi ustalony przedrostek 0:0:0:0:FFFF. Przykładowy odwzorowany adres IPv4 wygląda następująco:

```
::FFFF:192.1.1.1
```

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

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

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

Ograniczenia protokołu IPv6

W wersji V5R2 systemu i5/OS obsługa protokołu IPv6 jest do pewnego stopnia ograniczona. Poniższa tabela zawiera listę tych ograniczeń i wynikających z nich kwestii związanych z pisaniem aplikacji gniazd.

Tabela 11. Ograniczenia protokołu IPv6 i ich skutki

Ograniczenie	Skutek
Tabela hostów systemu iSeries nie obsługuje adresów IPv6.	Funkcje API getaddrinfo() i getnameinfo() nie są w stanie znaleźć adresów w tabeli hostów. Te funkcje odnajdują adresy jedynie w systemie nazw domen.
Funkcje API gethostbyname() i gethostbyaddr() obsługują jedynie rozstrzygnięcie adresów protokołu IPv4.	Jeśli wymagane jest rozstrzygnięcie adresów protokołu IPv6, należy użyć funkcji API getaddrinfo() i getnameinfo().

Odsyłacze pokrewne

“Wymagania wstępne dla programowania z użyciem gniazd” na stronie 2

Przed przystąpieniem do pisania aplikacji używającej gniazd należy wykonać następujące zadania:

“Scenariusze dla gniazd: tworzenie aplikacji komunikujących się z klientami IPv4 i IPv6” na stronie 74

Opis typowej sytuacji, w której użytkownik może użyć rodziny adresów AF_INET6.

Informacje pokrewne

Porównanie protokołów IPv4 i IPv6

recvfrom()

accept()

getpeername()

sendto()

connect()
bind()
gethostbyname()
getaddrinfo()
gethostbyaddr()
getnameinfo()

Używanie rodziny adresów AF_UNIX

Gniazda rodziny adresów AF_UNIX (gniazda używające rodzin adresów AF_UNIX lub AF_UNIX_CCSD) mogą być zorientowane na połączenie (typu SOCK_STREAM) lub bezpołączeniowe (typu SOCK_DGRAM).

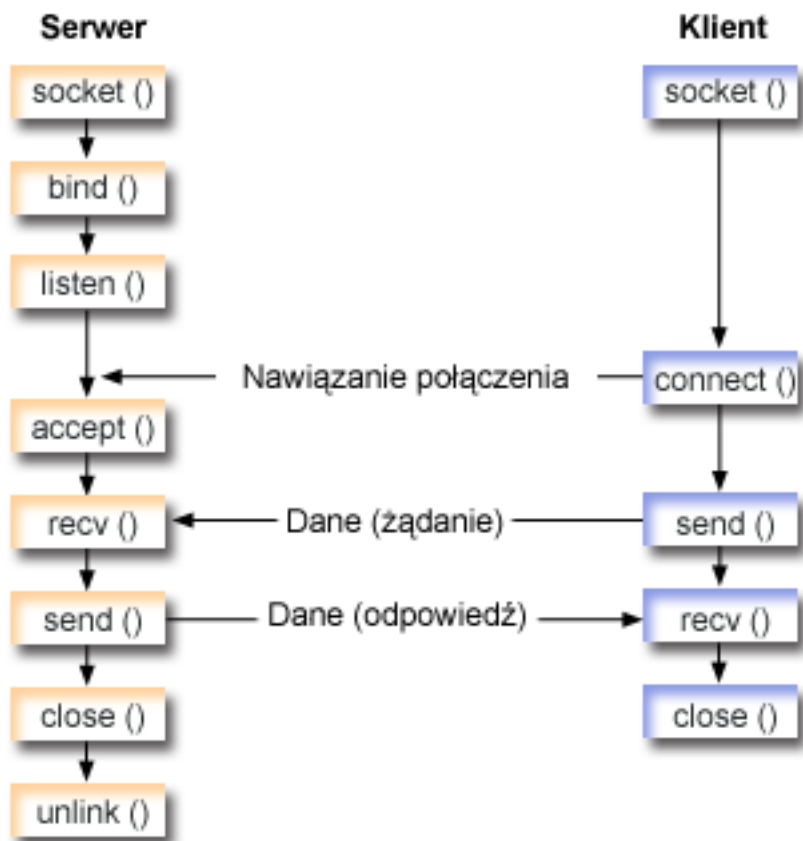
Ponieważ żadna zewnętrzna funkcja komunikacyjna nie łączy dwóch procesów, obydwa typy są tak samo niezawodne.

Gniazda datagramowe domeny UNIX działają inaczej niż gniazda datagramowe UDP. W przypadku gniazda datagramowego UDP program typu klient nie musi wywoływać funkcji bind(), gdyż system automatycznie przypisze numer nie używanego portu. Serwer może następnie wysłać datagram z powrotem na ten numer portu. Jednak w przypadku gniazd datagramowych domeny UNIX system nie przypisze automatycznie nazwy ścieżki dla klienta. Dlatego wszystkie programy typu klient korzystające z datagramów domeny UNIX mogą wywołać funkcję bind(). Dokładna nazwa ścieżki podana w funkcji bind() klienta jest wysyłana do serwera. Dlatego, jeśli klient określi względną ścieżkę (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 ścieżka, 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 w odniesieniu do bieżącego katalogu roboczego.

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

Poniższy rysunek ilustruje relację klient/serwer rodziny adresów AF_UNIX.



Przebieg zdarzeń w gnieździe: aplikacja serwera używająca rodziny adresów AF_UNIX

Pierwszy przykład wykorzystuje następującą sekwencję wywołań funkcji:

1. Funkcja `socket()` zwraca deskryptor gniazda odpowiadający punktowi końcowemu. Instrukcja ta informuje również, że dla tego gniazda zostanie użyta rodzina adresów UNIX z transportem strumieniowym (`SOCK_STREAM`). Funkcja ta zwraca deskryptor gniazda odpowiadający punktowi końcowemu. Można użyć także funkcji `socketpair()` aby zainicjować gniazdo UNIX.
AF_UNIX i AF_UNIX_CCSID to jedyne rodziny adresów, które obsługują funkcję `socketpair()`. Funkcja `socketpair()` zwraca dwa nienazwane i podłączone deskryptory gniazd.
2. Po utworzeniu deskryptora gniazda funkcja `bind()` pobiera unikalną nazwę gniazda.
Obszar nazwy gniazd domeny UNIX składa się z nazw ścieżek. Kiedy program używający gniazd wywołuje funkcję `bind()`, tworzona jest pozycja katalogu systemu plików. Jeśli ścieżka o podanej nazwie już istnieje, działanie funkcji `bind()` nie powodzi się. Dlatego, program używający gniazd domeny UNIX powinien zawsze wywołać funkcję `unlink()`, aby po zakończeniu działania usunąć tę pozycję katalogu.
3. Funkcja `listen()` umożliwia serwerowi przyjęcie połączenia przychodzącego od klienta. W tym przykładzie kolejka (backlog) ma wartość 10. Oznacza to, że system umieści w kolejce pierwsze 10 przychodzących żądań połączenia, a następne odrzuci.
4. Funkcja `recv()` odbiera dane z aplikacji klienta. W tym przykładzie klient wysła 250 bajtów danych. Na tej podstawie może użyć opcji gniazda `SO_RCVLOWAT` i określić, że funkcja `recv()` ma pozostać w uśpieniu do momentu nadejścia całych 250 bajtów danych.
5. Funkcja `send()` odsyła dane do klienta.
6. Funkcja `close()` zamyka wszystkie otwarte deskryptory gniazd.
7. Funkcja `unlink()` usuwa nazwę ścieżki UNIX z systemu plików.

Przebieg zdarzeń w gnieździe: aplikacja klienta używająca rodziny adresów AF_UNIX

Kolejny przykład wykorzystuje następującą sekwencję wywołań funkcji:

1. Funkcja `socket()` zwraca deskryptor gniazda odpowiadający punktowi końcowemu. Instrukcja ta informuje również, że dla tego gniazda zostanie użyta rodzina adresów UNIX z transportem strumieniowym (`SOCK_STREAM`). Funkcja ta zwraca deskryptor gniazda odpowiadający punktowi końcowemu. Można użyć także funkcji `socketpair()` aby zainicjować gniazdo UNIX.
AF_UNIX i AF_UNIX_CCSID to jedyne rodziny adresów, które obsługują funkcję `socketpair()`. Funkcja `socketpair()` zwraca dwa nienazwane i podłączone deskryptory gniazd.
2. Po odebraniu deskryptora gniazda należy użyć funkcji `connect()` do nawiązania połączenia z serwerem.
3. Funkcja `send()` wysyła 250 bajtów danych określonych w aplikacji serwera przez opcję gniazda `SO_RCVLOWAT`.
4. Funkcja `recv()` wykonuje wielokrotnie pętlę do momentu, gdy zostanie odebrane całe 250 bajtów.
5. Funkcja `close()` zamyka wszystkie otwarte deskryptory gniazd.

Odsyłacze pokrewne

“Rodzina adresów AF_UNIX” na stronie 10

Rodzina adresów AF_UNIX umożliwia komunikację międzyprocesową w ramach jednego systemu, w którym używane są funkcje API gniazd. Adres jest w rzeczywistości nazwą ścieżki do pozycji systemu plików.

“Wymagania wstępne dla programowania z użyciem gniazd” na stronie 2

Przed przystąpieniem do pisania aplikacji używającej gniazd należy wykonać następujące zadania:

“Używanie rodziny adresów AF_UNIX_CCSID” na stronie 36

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

Informacje pokrewne

`close()`
`socket()`
`bind()`
`unlink()`
`listen()`
`send()`
`recv()`
`socketpair()`
`connect()`

Przykład: aplikacja serwera używająca rodziny adresów AF_UNIX:

Poniższy przykład przedstawia program korzystający z rodziny adresów AF_UNIX. Rodzina adresów AF_UNIX używa wielu tych samych wywołań funkcji gniazd co inne rodziny adresów z tym, że do identyfikacji aplikacji serwera używa struktury nazwy ścieżki.

Uwaga: Korzystając z przykładowego kodu, użytkownik akceptuje warunki określone w temacie “Informacje dotyczące licencji na kod” na stronie 192.

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

/* **** */
/* Wymagane przez program pliki nagłówkowe. */
/* **** */
#include <stdio.h>
```

```

#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/un.h>

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

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

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

        /*****
        /* Po utworzeniu deskryptora gniazda funkcja bind() pobiera */
        /* unikalną nazwę gniazda. */
        *****/
        memset(&serveraddr, 0, sizeof(serveraddr));
        serveraddr.sun_family = AF_UNIX;
        strcpy(serveraddr.sun_path, SERVER_PATH);

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

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

```

```

    perror("Niepowodzenie listen()");
    break;
}

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

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

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

if (rc < 0)
{
    perror("Niepowodzenie setsockopt(SO_RCVLOWAT)");
    break;
}

/*****
/* Odebranie 250 bajtów od klienta */
*****/
rc = recv(sd2, buffer, sizeof(buffer), 0);
if (rc < 0)
{
    perror("Niepowodzenie recv()");
    break;
}
printf("Otrzymano dane, bajtów: %d\n", rc);
if (rc == 0 ||
    rc < sizeof(buffer))
{
    printf("Klient zamknął połączenie przed wysłaniem\n");
    printf("wszystkich danych\n");
    break;
}

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

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

} while (FALSE);

/*****

```

```

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

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

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

```

Przykład: aplikacja klienta używająca rodziny adresów AF_UNIX:

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

Poniższy program przykładowy używa rodziny adresów AF_UNIX do nawiązania połączenia klienta z serwerem.

Uwaga: Korzystając z przykładowego kodu, użytkownik akceptuje warunki określone w temacie “Informacje dotyczące licencji na kod” na stronie 192.

```

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

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

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

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

```



```

/*****/
/* Funkcja socket() zwraca deskryptor gniazda stanowiącego punkt */
/* końcowy. Instrukcja ta określa także, że dla tego gniazda */
/* użyta zostanie rodzina adresów UNIX_CCSID ze strumieniowym */
/* protokołem transportowym (SOCK_DGRAM). */
/*****/
sd = socket(AF_UNIX, SOCK_STREAM, 0);
if (sd < 0)
{
perror("Niepowodzenie socket()");
break;
}

/*****/
/* 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)
strncpy(serveraddr.sun_path, argv[1]);
else
strncpy(serveraddr.sun_path, SERVER_PATH);

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

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

/*****/
/* W tym przykładzie wiadomo, że serwer odpowie wysłaniem tych */
/* samych 250 bajtów, które 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 przysłanych w oddzielnych pakietach, */
/* funkcja recv() będzie wielokrotnie uruchamiana, dopóki */
/* wszystkie 250 bajtów nie zostanie odebrane. */
/*****/
bytesReceived = 0;
while (bytesReceived < BUFFER_LENGTH)
{
rc = recv(sd, & buffer[bytesReceived],
BUFFER_LENGTH - bytesReceived, 0);
if (rc < 0)
{

```

```

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

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

} while (FALSE);

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

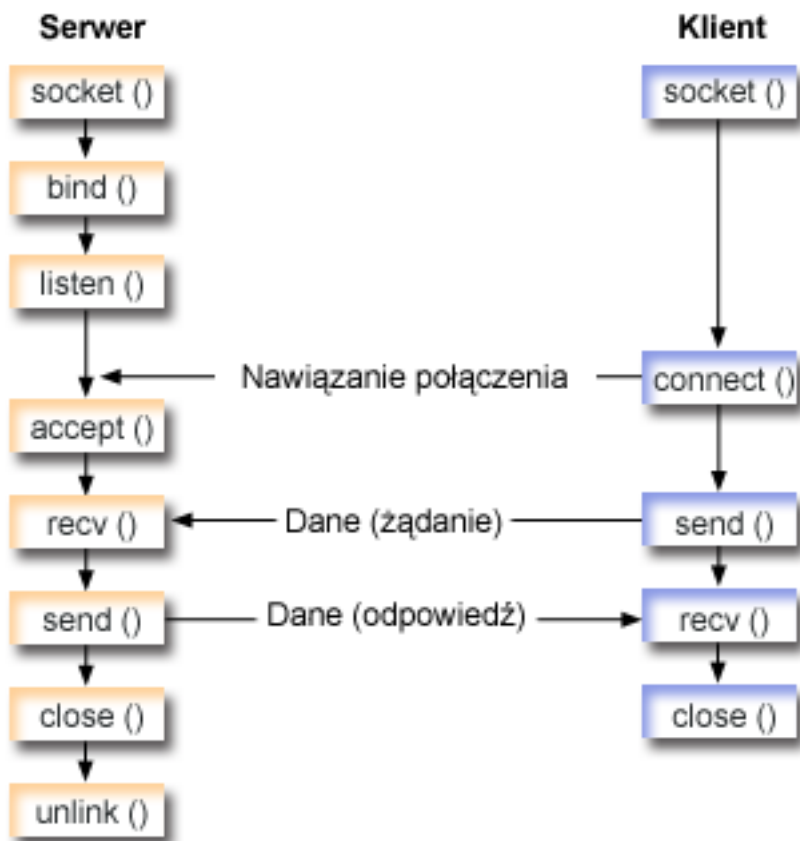
```

Używanie rodziny adresów AF_UNIX_CCSID

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

Aby rozpocząć pracę z aplikacją używającą gniazd AF_UNIX_CCSID, należy zapoznać się ze strukturą **Qlg_Path_Name_T**, w celu określenia formatu wyjściowego.

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



Przebieg zdarzeń w gnieździe: aplikacja serwera używająca rodziny adresów AF_UNIX_CCSID

Pierwszy przykład wykorzystuje następującą sekwencję wywołań funkcji:

1. Funkcja `socket()` zwraca deskryptor gniazda odpowiadający punktowi końcowemu. Instrukcja ta informuje również, że dla tego gniazda zostanie użyta rodzina adresów `UNIX_CCSID` z transportem strumieniowym (`SOCK_STREAM`). Można użyć także funkcji `socketpair()` aby zainicjować gniazdo UNIX.
`AF_UNIX` i `AF_UNIX_CCSID` to jedyne rodziny adresów, które obsługują funkcję `socketpair()`. Funkcja `socketpair()` zwraca dwa nienazwane i podłączone deskryptory gniazd.
2. Po utworzeniu deskryptora gniazda funkcja `bind()` pobiera unikalną nazwę gniazda.
 Obszar nazwy gniazda domeny UNIX składa się z nazw ścieżek. Kiedy program używający gniazda wywołuje funkcję `bind()`, tworzona jest pozycja katalogu systemu plików. Jeśli ścieżka o podanej nazwie już istnieje, działanie funkcji `bind()` nie powodzi się. Dlatego, program używający gniazda domeny UNIX powinien zawsze wywołać funkcję `unlink()`, aby po zakończeniu działania usunąć tę pozycję katalogu.
3. Funkcja `listen()` umożliwia serwerowi przyjęcie połączenia przychodzącego od klienta. W tym przykładzie kolejka (backlog) ma wartość 10. Oznacza to, że system umieści w kolejce pierwsze 10 przychodzących żądań połączenia, a następne odrzuci.
4. Serwer używa funkcji `accept()` do zaakceptowania żądania połączenia przychodzącego. Wywołanie funkcji `accept()` zostanie zablokowane na nieokreślony czas oczekiwania na połączenie przychodzące.
5. Funkcja `recv()` odbiera dane z aplikacji klienta. W tym przykładzie klient wysłała 250 bajtów danych. Na tej podstawie może użyć opcji gniazda `SO_RCVLOWAT` i określić, że funkcja `recv()` ma pozostać w uśpieniu do momentu nadejścia całych 250 bajtów danych.
6. Funkcja `send()` odsyła dane do klienta.
7. Funkcja `close()` zamyka wszystkie otwarte deskryptory gniazd.

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

Przebieg zdarzeń w gnieździe: aplikacja klienta używająca rodziny adresów AF_UNIX_CCSID

Kolejny przykład wykorzystuje następującą sekwencję wywołań funkcji:

1. Funkcja `socket()` zwraca deskryptor gniazda odpowiadający punktowi końcowemu. Instrukcja ta informuje również, że dla tego gniazda zostanie użyta rodzina adresów UNIX z transportem strumieniowym (`SOCK_STREAM`). Funkcja ta zwraca deskryptor gniazda odpowiadający punktowi końcowemu. Można użyć także funkcji `socketpair()` aby zainicjować gniazdo UNIX.
AF UNIX i AF_UNIX_CCSID to jedyne rodziny adresów, które obsługują funkcję `socketpair()`. Funkcja `socketpair()` zwraca dwa nienazwane i połączone deskryptory gniazd.
2. Po odebraniu deskryptora gniazda należy użyć funkcji `connect()` do nawiązania połączenia z serwerem.
3. Funkcja `send()` wysyła 250 bajtów danych określonych w aplikacji serwera przez opcję gniazda `SO_RCVLOWAT`.
4. Funkcja `recv()` wykonuje wielokrotnie pętlę do momentu, gdy zostanie odebrane całe 250 bajtów.
5. Funkcja `close()` zamyka wszystkie otwarte deskryptory gniazd.

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.

“Używanie rodziny adresów AF_UNIX” na stronie 29

Gniazda rodziny adresów AF_UNIX (gniazda używające rodzin adresów AF UNIX lub AF_UNIX_CCSID) mogą być zorientowane na połączenie (typu SOCK_STREAM) lub bezpołączeniowe (typu SOCK_DGRAM).

Informacje pokrewne

Format nazwy ścieżki

`recvfrom()`
`accept()`
`getpeername()`
`getsockname()`
`recvmsg()`
`close()`
`socket()`
`bind()`
`unlink()`
`listen()`
`send()`
`connect()`
`recv()`
`socketpair()`

Przykład: aplikacja serwera używająca rodziny adresów AF_UNIX_CCSID:

Poniższe programy przykładowe używają rodziny adresów AF_UNIX_CCSID.

Uwaga: Korzystając z przykładowego kodu, użytkownik akceptuje warunki określone w temacie “Informacje dotyczące licencji na kod” na stronie 192.

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

```

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

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

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

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

        /*****/
        /* Po utworzeniu deskryptora gniazda funkcja bind() pobiera */
        /* unikalną nazwę gniazda. */
        /*****/
        memset(&serveraddr, 0, sizeof(serveraddr));
        serveraddr.sunc_family    = AF_UNIX_CCSID;
        serveraddr.sunc_format    = SO_UNC_USE_QLG;
        serveraddr.sunc_qlg.CCSID = 500;
        serveraddr.sunc_qlg.Path_Type = QLG_PTR_SINGLE;
        serveraddr.sunc_qlg.Path_Length = strlen(SERVER_PATH);
        serveraddr.sunc_path.p_unix = SERVER_PATH;

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

        /*****/
        /* Funkcja listen() umożliwia serwerowi przyjęcie połączeń */

```

```

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

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

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

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

if (rc < 0)
{
    perror("Niepowodzenie setsockopt(SO_RCVLOWAT)");
    break;
}

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

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

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

```

```

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

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

} while (FALSE);

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

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

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

```

Przykład: aplikacja klienta używająca rodziny adresów AF_UNIX_CCSID:

Poniższe programy przykładowe używają rodziny adresów AF_UNIX_CCSID.

Uwaga: Korzystając z przykładowego kodu, użytkownik akceptuje warunki określone w temacie “Informacje dotyczące licencji na kod” na stronie 192.

```

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

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

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

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

```

```

struct sockaddr_unc serveraddr;

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

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

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

```



```

/*****
/* W tym przykładzie wiadomo, że serwer odpowie wysłaniem tych */
/* samych 250 bajtów, które 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 przysłanych w oddzielnych pakietach, */
/* funkcja recv() będzie wielokrotnie uruchamiana, dopóki */
/* wszystkie 250 bajtów nie zostanie odebrane. */
/*****
bytesReceived = 0;
while (bytesReceived < BUFFER_LENGTH)
{
    rc = recv(sd, & buffer[bytesReceived],
              BUFFER_LENGTH - bytesReceived, 0);
    if (rc < 0)
    {
        perror("Niepowodzenie recv()");
        break;
    }
    else if (rc == 0)
    {
        printf("Serwer zamknął połączenie\n");
        break;
    }
}

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

} while (FALSE);

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

```

Zaawansowane zagadnienia dotyczące gniazd

W poniższych sekcjach omówiono zaawansowane zagadnienia dotyczące gniazd, które wychodzą poza zakres ogólnych wiadomości o tym, czym są gniazda i jak działają. W sekcjach tych przedstawiono metody projektowania aplikacji używających gniazd dla większych i bardziej złożonych sieci.

Asynchroniczne operacje we/wy

Funkcje API asynchronicznych operacji we/wy udostępniają metodę dla wątków modelu klient serwer, umożliwiającą realizację współbieżnych operacji we/wy z efektywnym wykorzystaniem pamięci.

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

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

Zalety asynchronicznych operacji we/wy

- Asynchroniczne operacje we/wy zarządzają zasobami systemowymi w sposób bardziej efektywny. Kopiowanie danych z i do bufora użytkownika odbywa się asynchronicznie względem aplikacji inicjującej żądanie. To nakładające się przetwarzanie umożliwia efektywne wykorzystanie wielu procesorów i w wielu przypadkach poprawia szybkość stronicowania, ponieważ bufor systemowy są zwalniane do ponownego wykorzystania, kiedy nadsyłane są dane.
- 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 średnie obciążenie związane z uśpieniem i uaktywnieniem systemu.
- Asynchroniczne operacje we/wy wydajnie obsługują "aplikacje impulsowe".
- Asynchroniczne operacje we/wy oferują większą skalowalność.
- Asynchroniczne operacje we/wy oferują najbardziej efektywne metody obsługi przesyłania danych o dużej objętości. Opcja fillBuffer funkcji API QsoStartRecv() informuje system operacyjny o uzyskaniu dużych ilości danych przed zakończeniem asynchronicznych operacji we/wy. Duże ilości danych można również przysłać 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 beczynne 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 12. Funkcje API asynchronicznych operacji we/wy

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

Tabela 12. Funkcje API asynchronicznych operacji we/wy (kontynuacja)

Funckja	Opis
QsoDestroyIOCompletionPort()	Niszczy port we/wy zakończenia.
QsoWaitForIOCompletionPort()	Czeka na zakończone, nakładające się operacje we/wy. Port we/wy zakończenia reprezentuje ten port oczekiwania.
QsoStartAccept()	Rozpoczyna asynchroniczną operację akceptowania. Uwaga: Ta funkcja API obsługuje tylko gniazda typu SOCK_STREAM z rodziny adresów AF_INET i AF_INET6.
QsoStartRecv()	Rozpoczyna asynchroniczną operację odbioru. Uwaga: Ta funkcja API obsługuje tylko gniazda typu SOCK_STREAM z rodziny adresów AF_INET i AF_INET6.
QsoStartSend()	Rozpoczyna asynchroniczną operację wysyłania. Uwaga: Ta funkcja API obsługuje tylko gniazda typu SOCK_STREAM z rodziny adresów AF_INET i AF_INET6.
QsoPostIOCompletion()	Pozwala aplikacji powiadomić port zakończenia o wystąpieniu pewnej funkcji lub czynności.

Sposób działania asynchronicznych operacji we/wy

Aplikacja tworzy port we/wy zakończenia za pomocą funkcji API QsoCreateIOCompletionPort(). Funkcja ta zwraca uchwyt, którego można użyć do zaplanowania i oczekiwania na zakończenie żądań asynchronicznych operacji we/wy. Następnie aplikacja uruchamia funkcję operacji wejścia lub operacji wyjścia, podając uchwyt portu zakończenia operacji we/wy. Kiedy operacje we/wy zostają zakończone, informacje o statusie i zdefiniowany przez aplikację uchwyt zostają zapisane w podanym porcie we/wy zakończenia. Zapisanie do portu we/wy zakończenia uaktywnia dokładnie jeden z 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,
- zdefiniowany przez aplikację uchwyt, który został 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 zakończyła się, a uchwyt aplikacji został przekazany, wątek procesu roboczego określa następny krok w celu zakończenia połączenia klienta. Wątki procesów roboczych, które przetwarzają te zakończone operacje asynchroniczne, są w stanie obsłużyć wiele różnych żądań klienta i nie są powiązane tylko z jednym z nich. Ponieważ kopiowanie z i do buforów użytkowników odbywa się asynchronicznie względem procesów serwera, czasy oczekiwania dla żądań klientów 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 zademonstrowaną na 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 111

Aplikacja tworzy port we/wy zakończenia za pomocą funkcji API `QsoCreateIOCompletionPort()`.

“Zalecenia dotyczące projektowania aplikacji używających gniazd” na stronie 83

Aby projektować aplikacje używające gniazd, należy określić wymagania funkcjonalne, cele i potrzeby tej aplikacji.

“Przykłady: projekty aplikacji zorientowanych na połączenie” na stronie 86

W systemie iSeries istnieje wiele możliwości realizacji zorientowanego na połączenie serwera gniazd. Do utworzenia własnych programów zorientowanych na połączenie można użyć poniższych programów przykładowych.

“Przykład: używanie sygnałów z blokującymi funkcjami API gniazd” na stronie 163

Sygnały powiadają o zablokowaniu procesu lub aplikacji.

Informacje pokrewne

`gsk_secure_soc_startInit()`

`gsk_secure_soc_startRecv()`

`gsk_secure_soc_startSend()`

`QsoCreateIOCompletionPort()`

`QsoDestroyIOCompletionPort()`

`QsoWaitForIOCompletionPort()`

`QsoStartAccept()`


`QsoStartSend()`

`QsoStartRecv()`

`QsoPostIOCompletion()`

Gniazda chronione

Obecnie system i5/OS obsługuje dwa sposoby tworzenia aplikacji gniazd chronionych w systemie iSeries. Funkcje API `SSL_` i `Global Secure Toolkit (GSKit)` zapewniają prywatność komunikacji w otwartych sieciach komunikacyjnych, czyli najczęściej w Internecie.


Funkcje te umożliwiają aplikacjom typu klient/serwer komunikację zabezpieczoną przed podsłuchiwaniem, manipulowaniem i fałszowaniem danych. Obie funkcje obsługują uwierzytelnianie serwera i klienta oraz umożliwiają aplikacji użycie protokołu SSL (Secure Sockets Layer). Funkcje `GSKit` APIs są obsługiwane przez wszystkie platformy IBM  **server**, podczas gdy funkcje API SSL występują tylko w systemie operacyjnym i5/OS. Aby zapewnić współdziałanie pomiędzy platformami, podczas tworzenia aplikacji obsługujących połączenia gniazd chronionych zaleca się używanie funkcji API `GSKiT`.

Przegląd gniazd chronionych

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

Aplikacje obsługujące protokół SSL potrzebują zazwyczaj innego portu, niż aplikacje, które tego protokołu nie obsługują. Na przykład przeglądarka z obsługą SSL łączy się z serwerem protokołu Hypertext Transfer Protocol

(HTTP) obsługującym SSL za pomocą adresu Universal Resource Locator (URL), który zaczyna się od "HTTPS", a nie od "HTTP". W większości przypadków taki adres będzie wymagał otwarcia portu 443, a nie portu 80, używanego przez standardowy serwer HTTP.

Istnieje wiele zdefiniowanych wersji protokołu SSL. Najnowsza wersja, Transport Layer Security (TLS) wersja 1.0, stanowi ewolucyjną aktualizację protokołu SSL wersja 3.0. Zarówno funkcje API systemu iSeries - SSL_ - jak i funkcje API GSKit obsługują protokoły: TLS wersja 1.0, TLS wersja 1.0 kompatybilna z SSL wersja 3.0, SSL wersja 3.0, SSL wersja 2.0 i SSL wersja 3.0 kompatybilna z wersją 2.0. Szczegółowe informacje dotyczące TLS Wersja 1.0 znajdują się w dokumencie RFC 2246: "Transport Layer Security"  .

Funkcje API Global Secure ToolKit (GSKit)

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

Podobnie jak funkcje API SSL_, funkcje API GSKit umożliwiają zaimplementowanie protokołów SSL i TLS w aplikacji używającej gniazd. Jednakże funkcje API GSKit są obsługiwane przez wszystkie platformy IBM eServer i programuje się je łatwiej niż funkcje API SSL. Ponadto, nowe funkcje API GSKit APIs dodane zostały aby umożliwić możliwości asynchroniczne provide asynchronous capabilities for dla negocjowania bezpiecznej sesji, wysyłania i otrzymywania bezpiecznych danych. Niniejsze asynchroniczne funkcje API występują tylko w systemie i5/OS i nie mogą być przenoszone na inne platformy eServer.

Uwaga: Te funkcje API obsługuje tylko gniazda typu SOCK_STREAM z rodziny adresów AF_INET i AF_INET6.

Poniższa tabela przedstawia funkcje API GSKit:

Tabela 13. Funkcje API Global Secure Toolkit

Funkcja	Opis
Funkcja gsk_attribute_get_buffer()	Uzyskuje specyficzne informacje w postaci łańcucha znaków o bezpiecznej sesji lub o środowisku SSL, takie jak zbiór bazy certyfikatów, hasło bazy certyfikatów, identyfikator aplikacji i szyfr.
gsk_attribute_get_cert_info()	Uzyskuje specyficzne informacje o certyfikacie serwera lub klienta dla bezpiecznej sesji lub środowiska SSL.
gsk_attribute_get_enum_value()	Uzyskuje wartości specyficznych, wyszczególnionych danych dla bezpiecznej sesji lub dla środowiska SSL.
gsk_attribute_get_numeric_value()	Uzyskuje określone informacje numeryczne o bezpiecznej sesji lub środowisku SSL.
Funkcja gsk_attribute_set_callback()	Funkcja ta ustawia wskaźniki wywołań zwrotnych jako procedury w aplikacji użytkownika. Użytkownik może skorzystać z tych procedur dla szczególnych celów.
gsk_attribute_set_buffer()	Nadaje określonemu atrybutowi buforu wartość odpowiadającą określonej bezpiecznej sesji lub środowisku SSL.
gsk_attribute_set_enum()	Przypisuje wyszczególniony atrybut typu do wyszczególnionej wartości w bezpiecznej sesji lub środowisku SSL.
gsk_attribute_set_numeric_value()	Przypisuje określone informacje numeryczne bezpiecznej sesji lub środowisku SSL.
gsk_environment_close()	Zamyka środowisko SSL i zwalnia całą pamięć przypisaną środowisku.
gsk_environment_init()	Inicjuje środowisko SSL po ustawieniu wymaganych atrybutów.

Tabela 13. Funkcje API Global Secure Toolkit (kontynuacja)

Funkcja	Opis
<code>gsk_environment_open()</code>	Zwraca uchwyt środowiska SSL, który musi zostać zapisany i użyty podczas następnych wywołań funkcji <code>gsk</code> .
<code>gsk_secure_soc_close()</code>	Zamyka bezpieczną sesję i zwalnia wszystkie przypisane jej zasoby.
<code>gsk_secure_soc_init()</code>	Negocjuje bezpieczną sesję, używając zestawu atrybutów dla środowiska SSL i bezpiecznej sesji.
<code>gsk_secure_soc_misc()</code>	Realizuje różne funkcje bezpiecznej sesji.
<code>gsk_secure_soc_open()</code>	Uzyskuje pamięć dla bezpiecznej sesji, ustawia domyślne wartości atrybutów i zwraca uchwyt, który musi zostać zapisany i użyty podczas następnych wywołań funkcji związanych z bezpieczną sesją.
<code>gsk_secure_soc_read()</code>	Odbiera dane podczas bezpiecznej sesji.
<code>gsk_secure_soc_startInit()</code>	Uruchamia asynchroniczną negocjację bezpiecznej sesji, używając zestawu atrybutów dla środowiska SSL i bezpiecznej sesji.
<code>gsk_secure_soc_write()</code>	Zapisuje dane podczas bezpiecznej sesji.
<code>gsk_secure_soc_startRecv()</code>	Inicjuje asynchroniczną operację odbioru podczas bezpiecznej sesji.
<code>gsk_secure_soc_startSend()</code>	Inicjuje asynchroniczną operację wysłania podczas bezpiecznej sesji.
<code>gsk_strerror()</code>	Funkcja ta pobiera komunikat o błędzie i powiązany łańcuch tekstowy, który opisuje wartość zwracaną podczas wywołania funkcji API GSK.

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

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

Uwaga: Aby uzgadnianie SSL się powiodło, program serwera musi okazać certyfikat. Ponadto serwer musi mieć dostęp do klucza prywatnego powiązanego z certyfikatem serwera i zbioru bazy danych kluczy, w którym przechowywany jest certyfikat. W niektórych przypadkach klient także musi okazać certyfikat

podczas przetwarzania uzgadniania SSL. Dotyczy to sytuacji, w której serwer, z którym łączy się klient, ma włączone uwierzytelnianie klienta. Wywołania funkcji API `gsk_attribute_set_buffer(GSK_OS400_APPLICATION_ID)` lub `gsk_attribute_set_buffer(GSK_KEYRING_FILE)` identyfikują (różnymi metodami) zbiór bazy danych kluczy, z którego uzyskano klucz prywatny, i certyfikat użyte podczas uzgadniania.

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

Odsyłacze pokrewne

“Przykład: chroniony serwer GSKit z asynchronicznym odbieraniem danych” na stronie 118

Poniższego programu przykładowego można użyć do tworzenia chronionego serwera przy użyciu funkcji API Global Secure Toolkit (GSKit).

“Przykład: chroniony serwer GSKit z asynchronicznym uzgadnianiem” na stronie 127

Funkcja ta umożliwia tworzenie chronionych aplikacji serwera, które mogą obsługiwać żądania w sposób asynchroniczny.

“Przykład: chroniony klient używający funkcji API Global Secure Toolkit (GSKit)” na stronie 136

Poniższy kod stanowi przykład klienta używającego funkcji API GSKit.

Informacje pokrewne

`gsk_attribute_get_buffer()`
`gsk_attribute_get_cert_info()`
`gsk_attribute_get_enum_value()`
`gsk_attribute_get_numeric_value ()`
`gsk_attribute_set_callback()`
`gsk_attribute_set_buffer()`
`gsk_attribute_set_enum()`
`gsk_attribute_set_numeric_value()`
`gsk_environment_close()`
`gsk_environment_init()`
`gsk_environment_open()`
`gsk_secure_soc_close()`
`gsk_secure_soc_init()`
`gsk_secure_soc_misc()`
`gsk_secure_soc_open()`
`gsk_secure_soc_startInit()`
`gsk_secure_soc_read ()`
`gsk_secure_soc_write ()`
`gsk_secure_soc_startRecv()`
`gsk_secure_soc_startSend()`
`gsk_strerror()`
`socket()`
`bind()`
`connect()`
`listen()`
`accept()`
`close()`

Funkcje API SSL_

Funkcje API SSL_ umożliwiają tworzenie aplikacji gniazd chronionych w systemie iSeries.

W przeciwieństwie do funkcji API GSKit, funkcje API SSL_ występują tylko w systemie i5/OS. W poniższej tabeli opisano funkcje API SSL_ obsługiwane w implementacji systemu i5/OS implementation.

Tabela 14. Funkcje API SSL_

Funkcja	Opis
SSL_Create()	Umożliwia obsługę SSL dla określonego deskryptora gniazda.
SSL_Destroy()	Kończy obsługę SSL dla określonej sesji i gniazda SSL.
SSL_Handshake()	Inicjuje protokół uzgadniania SSL.
SSL_Init()	Inicjuje bieżące zadanie dla SSL i określa informacje o ochronie dla bieżącego zadania. Uwaga: Aby użyć protokołu SSL, wcześniej należy uruchomić funkcję API SSL_Init() lub SSL_Init_Application().
SSL_Init_Application()	Inicjuje bieżące zadanie dla SSL i określa informacje o ochronie dla bieżącego zadania. Uwaga: Aby użyć protokołu SSL, wcześniej należy uruchomić funkcję API SSL_Init() lub SSL_Init_Application().
SSL_Read()	Odbiera dane z deskryptora gniazda z obsługą SSL.
SSL_Write()	Zapisuje dane do deskryptora gniazda z obsługą SSL.
SSL_Sterror()	Odtwarza komunikaty o błędach wykonania SSL.
SSL_Perror()	Drukuje komunikaty o błędach SSL.
QlqSSL_Init()	Inicjuje bieżące zadanie dla SSL i określa informacje o ochronie dla bieżącego zadania, używając ścieżek nazw z obsługą narodowych wersji językowych.

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

- Wywołanie funkcji socket(), w celu uzyskania deskryptora gniazda.
- Wywołanie funkcji SSL_Init() lub SSL_Init_Application() w celu zainicjowania środowiska pracy dla przetwarzania SSL i w celu określenia informacji o ochronie SSL dla wszystkich sesji SSL, które będą uruchamiane w bieżącym zadaniu. Należy użyć tylko jednej z tych funkcji API. Zalecane jest użycie funkcji API SSL_Init_Application().
- Wywołania gniazd do uaktywnienia połączenia. Aplikacja wywołuje funkcję connect() w celu uaktywnienia połączenia z programem klienta lub funkcje bind(), listen() i accept() , aby umożliwić serwerowi akceptowanie przychodzących żądań połączeń.
- Klient wywołuje funkcję SSL_Create(), w celu włączenia obsługi SSL dla podłączonego gniazda.
- Wywołanie funkcji SSL_Handshake() w celu zainicjowania uzgadniania SSL parametrów szyfrujących.

Uwaga: Aby uzgadnianie SSL się powiodło, program serwera musi okazać certyfikat. Ponadto serwer musi mieć dostęp do klucza prywatnego powiązanego z certyfikatem serwera i zbioru bazy danych kluczy, w którym przechowywany jest certyfikat. W niektórych przypadkach klient także musi okazać certyfikat podczas przetwarzania uzgadniania SSL. Dotyczy to sytuacji, w której serwer, z którym łączy się klient, ma włączone uwierzytelnianie klienta. Funkcje API SSL_Init() i SSL_Init_Application() identyfikują (choć w różny sposób) zbiór bazy danych, z którego pobierany jest certyfikat, i klucz prywatny, używane podczas uzgadniania.

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

Odsyłacze pokrewne

“Przykład: chroniony serwer używający funkcji API SSL_” na stronie 141

Tworząc aplikacje chronione można oprócz funkcji API GSKit używać funkcji API SSL_. Funkcje API SSL_ APIs występują tylko w systemie operacyjnym i5/OS.

“Przykład: chroniony klient używający funkcji API SSL_” na stronie 145

Poniższy przykład to aplikacja klienta używająca funkcji API SSL_ do komunikowania się z aplikacją serwera używającą funkcji API SSL_:

Informacje pokrewne

socket()
listen()
bind()
connect()
accept()
close()
SSL_Create()
SSL_Destroy()
SSL_Handshake()
SSL_Init()
SSL_Init_Application()
SSL_Read()
SSL_Write()
SSL_Strerror()
SSL_Perror()

Komunikaty z kodami błędów funkcji API gniazd chronionych

Aby uzyskać informacje o komunikatach kodów błędów gniazda chronionego, należy postępować zgodnie ze wskazówkami.

1. W wierszu komend wpisz:

```
DSPMSGD RANGE(XXXXXXX)
```

gdzie XXXXXXXX jest identyfikatorem komunikatu dla danego kodu powrotu. Na przykład, jeśli kod powrotu był równy 3, wpisz:

```
DSPMSGD RANGE(CPDBC9)
```

2. Wybierz 1, aby wyświetlić tekst komunikatu.

Tabela 15. Komunikaty z kodami błędów funkcji API gniazd chronionych

Kod powrotu	ID komunikatu	Nazwa stałej
0	CPCBC80	GSK_OK
1	CPBCA1	GSK_INVALID_HANDLE
2	CPDBC3	GSK_API_NOT_AVAILABLE
3	CPDBC9	GSK_INTERNAL_ERROR
4	CPC3460	GSK_INSUFFICIENT_STORAGE
5	CPDBC95	GSK_INVALID_STATE
107	CPDBC98	GSK_KEYFILE_CERT_EXPIRED
201	CPBCA4	GSK_NO_KEYFILE_PASSWORD
202	CPDBC5	GSK_KEYRING_OPEN_ERROR
301	CPBCA5	GSK_CLOSE_FAILED
402	CPDBC81	GSK_ERROR_NO_CIPHERS

Tabela 15. Komunikaty z kodami błędów funkcji API gniazd chronionych (kontynuacja)

Kod powrotu	ID komunikatu	Nazwa stałej
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	CPDBC8B7	GSK_ERROR_BAD_V2_CIPHER
422	CPDBC8B7	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	CPDBC8B1	GSK_OS400_ERROR_PASSWORD_EXPIRED
6002	CPDBC9C9	GSK_OS400_ERROR_NOT_REGISTERED
6003	CPDBCAD	GSK_OS400_ERROR_NO_ACCESS
6004	CPDBC8B8	GSK_OS400_ERROR_CLOSED
6005	CPDBC8CB	GSK_OS400_ERROR_NO_CERTIFICATE_AUTHORITIES
6007	CPDBC8B4	GSK_OS400_ERROR_NO_INITIALIZE
6008	CPDBC8AE	GSK_OS400_ERROR_ALREADY_SECURE
6009	CPDBC8AF	GSK_OS400_ERROR_NOT_TCP
6010	CPDBC9C	GSK_OS400_ERROR_INVALID_POINTER
6011	CPDBC9B	GSK_OS400_ERROR_TIMED_OUT
6012	CPC8CBA	GSK_OS400_ASYNCHRONOUS_RECV
6013	CPC8CBB	GSK_OS400_ASYNCHRONOUS_SEND
6014	CPDBC8BC	GSK_OS400_ERROR_INVALID_OVERLAPPEDIO_T
6015	CPDBC8BD	GSK_OS400_ERROR_INVALID_IOCTLCOMPLETIONPORT

Tabela 15. Komunikaty z kodami błędów funkcji API gniazd chronionych (kontynuacja)

Kod powrotu	ID komunikatu	Nazwa stałej
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	CPDBCBI	SSL_ERROR_KEYPASSWORD_EXPIRED
-92	CPDBCBI	SSL_ERROR_CERTIFICATE_REJECTED
-93	CPDBCBI	SSL_ERROR_SSL_NOT_AVAILABLE
-94	CPDBCBI	SSL_ERROR_NO_INIT
-95	CPDBCBI	SSL_ERROR_NO_KEYRING
-97	CPDBCBI	SSL_ERROR_BAD_CIPHER_SUITE
-98	CPDBCBI	SSL_ERROR_CLOSED
-99	CPDBCBI	SSL_ERROR_UNKNOWN
-1009	CPDBCC9	SSL_ERROR_NOT_REGISTERED
-1011	CPDBCCB	SSL_ERROR_NO_CERTIFICATE_AUTHORITIES
-9998	CPBCD8	SSL_ERROR_NO_REUSE

Odsyłacze pokrewne

“Przykład: nawiązywanie chronionych połączeń” na stronie 118

Chroniony serwer i klienta można utworzyć przy użyciu funkcji API Global Secure ToolKit (GSKit) lub SSL_.

Obsługa klienta SOCKS

W serwerze iSeries zastosowano mechanizm SOCKS wersja 4, aby umożliwić programom, które używają rodziny adresów AF_INET z gniazdami typu SOCK_STREAM, komunikowanie się z programami serwera działającymi w systemach poza firewallem.

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

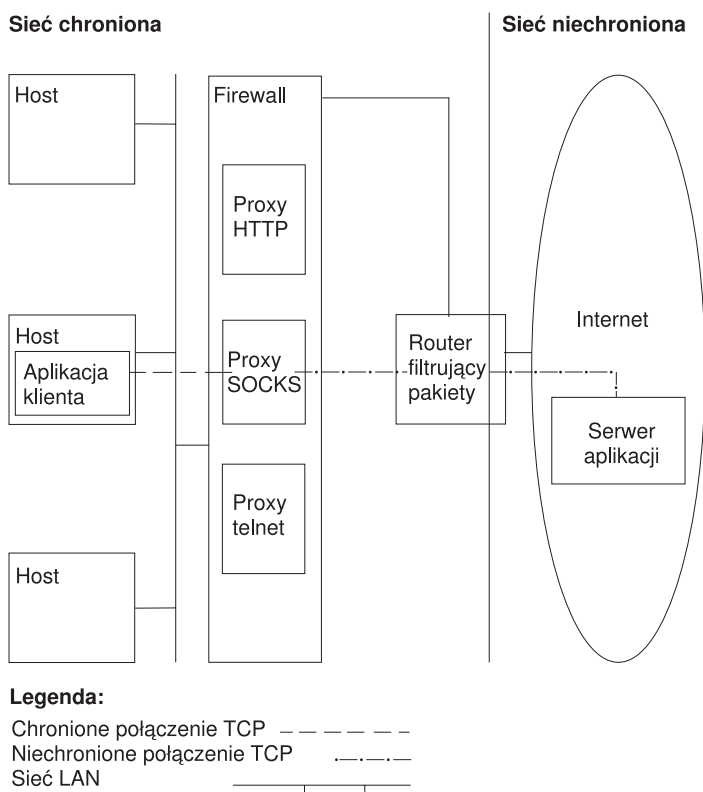
Aplikacje działające na serwerze w chronionej sieci wewnętrznej muszą wysyłać swoje zgłoszenia przez serwery proxy firewall. Serwery te mogą następnie przekazać zgłoszenia do serwera istniejącego w mniej chronionej sieci. Dodatkowo mogą przekazać odpowiedź do aplikacji serwera rozpoczynającego komunikację.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

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

Serwer proxy HTTP obsługuje tylko klientów HTTP.

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

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



RV4W201-01

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

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

Aby skonfigurować obsługę klienta SOCKS, wykonaj poniższe czynności:

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

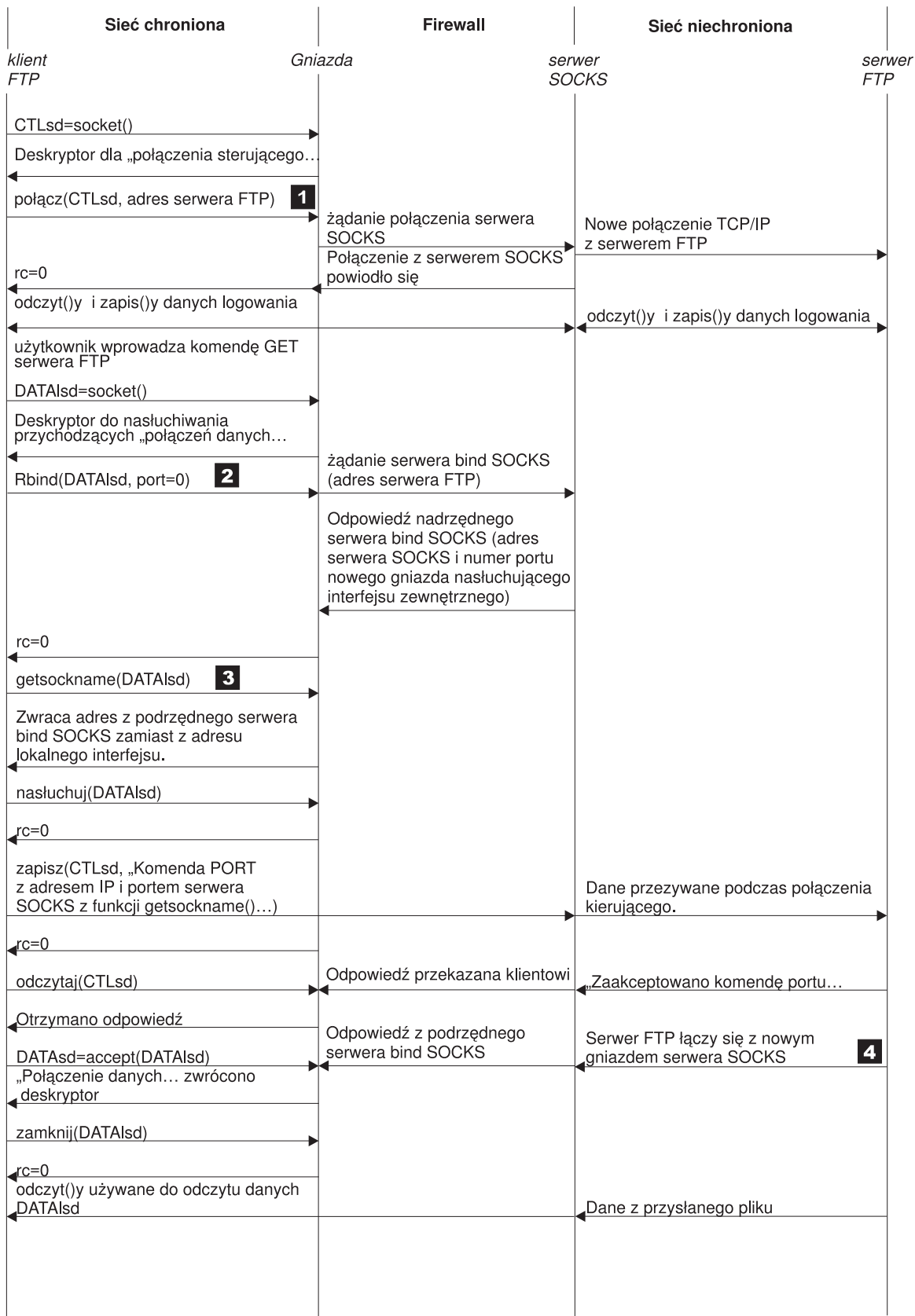
Uwaga: System zapisze dane o konfiguracji chronionego klienta SOCKS w zbiorze QASOSCFG w bibliotece QUSRSYS w systemie hosta chronionego klienta.

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

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

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

Poniższy rysunek przedstawia szczegółowy przegląd interakcji funkcji gniazd z serwerem SOCKS w sposób przezroczysty dla aplikacji. Klient FTP używa funkcji Rbind() zamiast funkcji bind(), ponieważ protokół FTP pozwala serwerowi FTP na nawiązywanie połączeń danych na żądanie wysłania plików lub danych z klienta FTP. Wywołuje ją dzięki ponownej kompilacji kodu klienta FTP z definicją preprocesora (#define) __Rbind, która definiuje funkcję bind() jako Rbind(). Alternatywnie aplikacja może mieć jawnie zakodowane Rbind() w odpowiednim kodzie źródłowym. Jeśli aplikacja nie wymaga połączeń przychodzących poprzez serwer SOCKS, nie należy używać funkcji Rbind().



RV4W200-1

Uwagi:

1. Klient FTP inicjuje połączenie wychodzące TCP do mniej chronionej sieci poprzez serwer SOCKS. Adres docelowy podany przez klienta FTP podczas wywołania funkcji connect to adres IP i port serwera FTP znajdującego się w sieci niechronionej. System chronionego hosta jest konfigurowany na zakładce SOCKS do kierowania tego połączenia przez serwer SOCKS. Po skonfigurowaniu system automatycznie przekieruje połączenia do serwera SOCKS podanego na zakładce SOCKS.
2. Gniazdo jest otwierane i wywoływana jest funkcja Rbind() w celu nawiązania połączenia przychodzącego TCP. To połączenie pochodzi z tego samego adresu IP, który został podany powyżej. Połączenia przychodzące i wychodzące przez serwer SOCKS dla określonego wątku muszą być nawiązywane parami. Innymi słowy, wszystkie połączenia przychodzące Rbind() muszą być nawiązywane bezpośrednio po połączeniu wychodzącym przez serwer SOCKS. 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 serwer SOCKS. W przykładzie adres jest wysyłany przez "połączenie kontrolne" Socket CtlSd do serwera FTP umieszczonego w mniej chronionej sieci. Jest to adres, z którym łączy się serwer FTP. Serwer FTP łączy się z serwerem SOCKS, a nie bezpośrednio z chronionym serwerem.
4. Serwer SOCKS ustanawia połączenie dla danych z klientem FTP i przekazuje dane pomiędzy klientem a serwerem FTP. Wiele serwerów SOCKS zezwala na połączenia z chronionym klientem w ustalonym okresie czasu. Jeśli serwer nie połączy się w tym okresie, dla funkcji accept() wystąpi błąd ECONNABORTED.

Informacje pokrewne

bind()

connect()

accept()

getsockname()

Rbind()

Ochrona wątków

Funkcja realizuje ochronę wątków, jeśli można ją uruchomić jednocześnie w wielu wątkach w ramach tego samego procesu.

Funkcja realizuje ochronę wątków wtedy i tylko wtedy, gdy wszystkie funkcje, które ta funkcja wywołuje, również realizują ochronę wątków. Funkcje API gniazd składają się z funkcji systemowych i sieciowych, które realizują ochronę wątków.

Wszystkie funkcje sieciowe, których nazwy kończą się przyrostkiem "_r", mają podobną semantykę i również realizują ochronę wątków.

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

Odsyłacze pokrewne

“Przykład: procedury sieciowe obsługujące ochronę wątków i używające funkcji gethostbyaddr_r()” na stronie 148
Poniżej przedstawiono przykład programu, w którym wykorzystano funkcję gethostbyaddr_r().

“Przykład: odpytywanie i aktualizacja serwera DNS” na stronie 171

Poniższy przykład ilustruje, jak wysyłać zapytania do systemu nazw domen (DNS) i jak aktualizować jego rekordy.

Nieblokujące operacje we/wy

Gdy aplikacja wywoła jedną z funkcji wejściowych gniazda, a nie będzie żadnych danych do odczytania, funkcja zablokuje się i nie zakończy się, dopóki nie danych do odczytania.

Podobnie aplikacja może zablokować funkcję wyjściową gniazda, jeśli nie będzie mogła natychmiast wysłać danych. Funkcje connect() oraz accept() mogą również zablokować się podczas oczekiwania na ustanowienie połączenia z innymi programami.

l Gniazda udostępniają aplikacjom metody umożliwiające wywołanie funkcji, które się blokują w taki sposób, że
l mogą one być zakończone bez opóźnienia. Polegają one na wywołaniu funkcji fcntl(), która włącza opcję
l **O_NONBLOCK**, lub funkcji ioctl(), która włącza opcję **FIONBIO**. Po uruchomieniu trybu nieblokującego, jeśli
l funkcja nie może być zakończona bez zablokowania, zostaje ona zakończona natychmiast. Funkcja connect() może
l zwrócić parametr [EINPROGRESS], który oznacza, że inicjowanie połączenia zostało uruchomione. Następnie można
l wybrać funkcję poll() lub select() w celu stwierdzenia kiedy zakończono połączenie. Dla wszystkich innych funkcji, na
l które praca w trybie nieblokującym może mieć wpływ, kod błędu [EWOULDBLOCK] wskazuje, że wywołanie się nie
l powiodło.

Trybu nieblokującego można użyć z następującymi funkcjami gniazd:

- accept()
- connect()
- gsk_secure_soc_read()
- gsk_secure_soc_write()
- read()
- readv()
- recv()
- recvfrom()
- recvmsg()
- send()
- send_file()
- send_file64()
- sendmsg()
- sendto()
- SSL_Read()
- SSL_Write()
- write()
- writev()

Odsyłacze pokrewne

“Przykład: nieblokujące operacje we/wy i funkcja select()” na stronie 151

Poniższy program przykładowy wykorzystuje nieblokujące operacje we/wy i funkcję API select().

Informacje pokrewne

fcntl()

accept()

ioctl()

recv()

send()

connect()

gsk_secure_soc_read ()

gsk_secure_soc_write ()

SSL_Read()

SSL_Write()

read()

readv()


```
recvfrom()
recvmsg()
send_file()
send_file64()
sendmsg()
sendto()
write()
writev()
```

Sygnały

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

Są dwa sygnały asynchroniczne, które gniazda wysyłają do aplikacji.

1. *SIGURG* to sygnał, wysyłany gdy zostaną odebrane dane typu out-of-band (OOB) przez gniazdo obsługujące dane OOB. Na przykład, gniazdo typu SOCK STREAM z rodziny adresów AF_INET można skonfigurować, aby wysyłało sygnał.
2. *SIGIO* to sygnał wysyłany, gdy w gnieździe dowolnego typu zostaną odebrane normalne dane, dane typu OOB, wystąpią warunki błędów lub zdarzy się cokolwiek innego.

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

Aplikacja żąda od systemu wysyłanie sygnału *SIGURG* jedną z następujących metod:

- poprzez wywołanie funkcji `fcntl()` oraz podanie identyfikatora procesu lub grupy procesu komendy `F_SETOWN`.
- poprzez wywołanie funkcji `ioctl()` oraz określenie komendy (żądania) `FIOSETOWN` albo `SIOCSPGRP`.

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

- wywołanie funkcji `fcntl()` i określenie komendy `F_SETFL` ze znacznikiem binarnym `FASYNC`.
- wywołanie funkcji `ioctl()` i określenie komendy `FIOASYNC`.

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

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

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

- `accept()`
- `connect()`

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

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

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 163

Sygnały powiadamiają o zablokowaniu procesu lub aplikacji.

Informacje pokrewne

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

Rozsyłanie grupowe IP

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

- | Hosty należące do grupy mogą znajdować się w tej samej podsieci lub w różnych podsieciach połączonych routerami
- | obsługującymi rozsyłanie grupowe. Hosty mogą dołączać do grup i opuszczać je w każdej chwili. Nie ma ograniczeń
- | dotyczących położenia ani liczby członków grupy hostów. Grupę hostów AF_INET identyfikuje klasa D adresów
- | internetowych w zakresie od 224.0.0.1 do 239.255.255.255. Dla adresów rodziny AF_INET6, adres IPv6 zaczynający
- | się na FF00::/8 identyfikowany jest jako adres rozsyłania grupowego. Szczegółowe informacje na ten temat znajdują
- | się w sekcji RFC 3513: "Architektura Adresowania Protokołu Internetowego v6 (IPv6).

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

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

| Funkcja setsockopt() akceptuje następujące opcje poziomu IPPROTO_IP:

- | • IPV6_MULTICAST_IF: Konfiguruje interfejs, poprzez który wysyłane są wychodzące datagramy rozsyłania
| grupowego.
- | • IPV6_MULTICAST_HOPS: Określa wartości limitu przeskoku używane do późniejszego rozsyłania grupowego
| pakietów przez gniazdo.
- | • IPV6_MULTICAST_LOOP: Określa, czy kopia wychodzącego datagramu rozsyłania ma być dostarczana do hosta
| wysyłającego, który jest członkiem grupy rozsyłania.
- | • IPV6_JOIN_GROUP: Dołącza do podanej grupy rozsyłania grupowego.
- | • IPV6_LEAVE_GROUP: Opuszcza podaną grupę rozsyłania grupowego.

Odsyłacze pokrewne

“Przykłady: użycie rozsyłania grupowego z obsługą AF_INET” na stronie 166

Rozsyłanie grupowe IP zapewnia aplikacjom możliwość wysyłania jednego datagramu IP, który odbierany jest przez grupę hostów w sieci.

Informacje pokrewne

setsockopt()

Przesyłanie danych pliku - send_file() i accept_and_recv()

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

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

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

Funkcja accept_and_recv() jest kombinacją funkcji: accept(), getsockname(), and recv().

Odsyłacze pokrewne

“Przykład: przesyłanie danych za pomocą funkcji send_file() i accept_and_recv()” na stronie 176

Poniższe przykłady umożliwiają serwerowi komunikację z klientem za pomocą funkcji send_file() i accept_and_recv().

Informacje pokrewne

send_file()

accept_and_recv()

Dane spoza pasma

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

Dane strumieniowe są zwykle odbierane w takiej samej kolejności, w jakiej zostały wysłane. Dane OOB są otrzymywane niezależnie od ich pozycji w strumieniu (niezależnie od kolejności, w jakiej zostały wysłane). Dzieje się tak, ponieważ podczas przesyłania tak oznaczonych danych z programu A do programu B, program B jest powiadamiany o ich przyjściu.

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

Dane OOB są wysyłane poprzez określenie opcji MSG_OOB w funkcji send(), sendto(), i sendmsg().

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

Po stronie odbierającej jest to bardziej skomplikowane:

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

Uwaga: Wartość wskazująca, który bajt wskazuje znacznik OOB, jest ustawiana globalnie dla całego systemu (wszystkie aplikacje korzystają z tej wartości). Wartość ta musi być spójna pomiędzy lokalnym i zdalnym końcem połączenia TCP. Musi być używana zgodnie pomiędzy aplikacjami typu klient i typu serwer.

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

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

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

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

Jeśli zostanie ustawiona opcja gniazda SO_OOBINLINE, wtedy wszystkie wysłane dane są przechowywane w normalnym strumieniu danych. Dane można odtworzyć przez wywołanie jednej z trzech funkcji odbierających, bez określania opcji MSG_OOB (jeśli zostanie ona określona, zostanie zwrócony błąd [EINVAL]). Jeśli wysłanych zostanie wiele wystąpień danych OOB, to nie zostaną one utracone.

- Dane OOB nie są odrzucane, jeśli nie zostanie ustawiona opcja SO_OOBINLINE; zostają one odebrane, a następnie użytkownik włącza opcję SO_OOBINLINE. Początkowy bajt OOB jest uważany za normalne dane.
- Jeśli opcja SO_OOBINLINE nie została ustawiona, dane OOB zostały wysłane, a program odbierający wywołał funkcję wejścia do odbioru danych OOB, wtedy znacznik będzie w dalszym ciągu poprawny. Program odbierający może w dalszym ciągu sprawdzić, czy odczytany wskaźnik wskazuje znacznik OOB, nawet jeśli bajt OOB został odczytany.

Pojęcia pokrewne

“Sygnały” na stronie 59

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

Informacje pokrewne

sendmsg()

Komenda zmiany atrybutów protokołu TCP (CHGTCPA)

Multipleksowanie operacji we/wy - funkcja select()

Asynchroniczne operacje we/wy zapewniają znacznie efektywniejszą metodę maksymalizowania zasobów aplikacji niż funkcja API select(), dlatego zalecane jest używanie tych operacji. Jednak konkretny projekt aplikacji może dopuszczać użycie funkcji select().

Podobnie jak asynchroniczne operacje we/wy funkcja select() tworzy wspólny punkt jednoczesnego oczekiwania dla wielu warunków. Jednak funkcja select() umożliwia aplikacji określenie zestawu deskryptorów w celu:

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

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

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

Odsyłacze pokrewne

“Przykład: nieblokujące operacje we/wy i funkcja select()” na stronie 151

Poniższy program przykładowy 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 możliwy jest poprzez nazwę, adres lub poprzez dostęp sekwencyjny. Te funkcje sieciowe (lub procedury) są wymagane podczas konfigurowania komunikacji pomiędzy programami, które działają w sieci, i nie są używane przez gniazda rodziny AF_UNIX.

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 adresach sieciowych Internetu,
- przekształcają adresy IP i notacje w postaci dziesiętnej.

W skład procedur sieciowych wchodzi tzw. procedury tłumaczące. Służą one do tworzenia, wysyłania i interpretowania pakietów dla serwerów nazw w domenie internetowej oraz do tłumaczenia nazw. Procedury tłumaczące są zwykle wywoływane przez funkcje gethostbyname(), gethostbyaddr(), getnameinfo() i getaddrinfo(), ale mogą być również wywoływane bezpośrednio. Procedury tłumaczące są używane głównie przez aplikacje używające gniazd do dostępu do systemu nazw domen (DNS).

Pojęcia pokrewne

“Charakterystyka gniazd” na stronie 6

Gniazda mają następujące cechy wspólne.

Odsyłacze pokrewne

“Przykład: procedury sieciowe obsługujące ochronę wątków i używające funkcji `gethostbyaddr_r()`” na stronie 148
Poniżej przedstawiono przykład programu, w którym wykorzystano funkcję `gethostbyaddr_r()`.

“System nazw domen”

Serwer iSeries zapewnia aplikacjom dostęp do systemu nazw domen (DNS) poprzez funkcje tłumaczące.

Informacje pokrewne

Funkcje sieciowe gniazd (procedury)

`gethostbyname()`

`getaddrinfo()`

`gethostbyaddr()`

`getnameinfo()`

System nazw domen

Serwer iSeries zapewnia aplikacjom dostęp do systemu nazw domen (DNS) poprzez funkcje tłumaczące.

Na system DNS składają się trzy główne komponenty:

- **Obszar nazwy domeny i zbiory danych** Określają one przestrzeń nazw o strukturze drzewa oraz dane związane z nazwami.
- **Serwery nazw** Są to programy serwera, które przechowują informacje o strukturze drzewa domeny.
- **Programy tłumaczące** Są to programy, które w odpowiedzi na żądania klientów wyodrębniają informacje z serwerów nazw.

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

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

- RFC 1034, "Domain names - concepts and facilities" (Nazwy domen - koncepcje i narzędzia)
- RFC 1035, "Domain names - implementation and specification" (Nazwy domen - implementacja i specyfikacja)
- RFC 1886, "DNS Extensions to support IP version 6" (Rozszerzenia DNS obsługujące protokół IP wersja 6)
- RFC 2136, "Dynamic Updates in the Domain Name System (DNS UPDATE)" (Dynamiczne aktualizacje systemu nazw domen - DNS UPDATE)
- RFC 2181, "Clarifications to the DNS Specification" (Objaśnienia specyfikacji DNS)
- RFC 2845, "Secret Key Transaction Authentication for DNS (TSIG)" (Uwierzytelnianie DNS za pomocą klucza tajnego - TSIG)
- RFC 3152, "DNS Delegation of IP6.ARPA" (Przeniesienie DNS do adresów IP6.ARPA)

Odsyłacze pokrewne

“Funkcje sieciowe gniazd” na stronie 63

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

Informacje pokrewne

DNS

Funkcje sieciowe gniazd (procedury)

Zmienne środowiskowe

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

Zmienne środowiskowe są sprawdzane tylko po zakończonym powodzeniu wywołaniu funkcji `res_init()` lub `res_ninit()`. Zatem, jeśli struktura została zainicjowana ręcznie, zmienne środowiskowe są ignorowane. Należy także zwrócić uwagę, iż struktura jest inicjowana tylko raz, dlatego późniejsze modyfikacje zmiennych środowiskowych również będą ignorowane.

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

LOCALDOMAIN

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

RES_OPTIONS

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

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

QIBM_BIND_RESOLVER_FLAGS

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

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

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

lub

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

QIBM_BIND_RESOLVER_SORTLIST

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

Informacje pokrewne

res_init()

res_ninit()

res_query()

Buforowanie danych

Buforowanie danych odpowiedzi na zapytania do serwera DNS jest wykonywane przez gniazda systemu i5/OS w celu zmniejszenia ruchu w sieci. Pamięć podręczna jest dodawana i aktualizowana, gdy jest to wymagane.

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

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

Odsyłacze pokrewne

“Przykład: odpytywanie i aktualizacja serwera DNS” na stronie 171

Poniższy przykład ilustruje, jak wysyłać zapytania do systemu nazw domen (DNS) i jak aktualizować jego rekordy.

Zgodność z Berkeley Software Distribution (BSD)

Gniazda są interfejsem systemu (BSD).

Semantyka, taka jak kody powrotu otrzymywane przez aplikację i argumenty dostępne w obsługiwanych funkcjach, należy do systemu BSD. Niektóre reguły semantyki BSD nie są dostępne w implementacji i5/OS i aby uruchomić w systemie typową aplikację opartą na gniazdach BSD, mogą być potrzebne zmiany.

Poniższa lista podsumowuje różnice pomiędzy implementacjami i5/OS i BSD.

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

/etc/hosts, /etc/usługi, /etc/sieci, and /etc/protokoły

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

/etc/resolv.conf

Implementacja i5/OS wymaga, aby te informacje zostały skonfigurowane za pomocą zakładki właściwości TCP/IP w programie iSeries Navigator. W celu użycia zakładki właściwości TCP/IP, wykonaj następujące czynności:

1. W programie iSeries Navigator, rozwiń **Serwer iSeries** → **Sieć** → **Konfiguracja TCP/IP**.
2. Kliknij prawym klawiszem myszy **Konfiguracja TCP/IP**.

3. Kliknij **Właściwości**.

bind()

W systemie BSD klient może utworzyć gniazdo AF UNIX za pomocą funkcji `socket()`, połączyć się z serwerem za pomocą funkcji `connect()`, a następnie powiązać nazwę z gniazdem za pomocą funkcji `bind()`. Implementacja i5/OS nie obsługuje tego scenariusza (funkcja `bind()` nie powiedzie się).

Funkcja close()

Implementacja i5/OS obsługuje licznik czasu zwłoki dla funkcji `close()`, za wyjątkiem gniazd AF INET przez SNA (Architektura Systemów Sieciowych). Niektóre implementacje BSD nie obsługują licznika czasu zwłoki dla funkcji `close()`.

connect()

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

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

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 zero i nieokreśloną strukturę adresu. Implementacja i5/OS BSD 4.4/ UNIX i inne mogą zwrócić małą strukturę adresu zawierającą tylko określenie rodziny adresów.

ioctl()

- W systemie BSD dla gniazd typu SOCK_DGRAM żądanie FIONREAD zwróci długość danych plus długość adresu. W implementacji 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()`.

listen()

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

Dane Out-of-band (OOB)

W implementacji i5/OS, dane OOB nie są odrzucane jeśli opcja `SO_OOBINLINE` została ustawiona, dane OOB zostają odebrane, a następnie użytkownik włącza opcję `SO_OOBINLINE`. Początkowy bajt OOB jest uważany za normalne dane.

parametr protokołu funkcji socket()

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

Funkcje res_xlate() i res_close()

Funkcje te wbudowane są w procedury tłumaczące implementacji i5/OS. Funkcja `res_xlate()` tłumaczy pakiety systemu nazw domen z kodu EBCDIC na kod ASCII oraz z kodu ASCII na kod EBCDIC. Funkcja `res_close()` jest używana do zamknięcia gniazda, które zostało użyte przez funkcję `res_send()` z ustawioną opcją `RES_STAYOPEN`. Powoduje to także zresetowanie struktury `_res`.

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 otrzymywane jest potwierdzenie dla danych wysyłanych przez operację wyjściową. Implementacja gniazd systemu i5/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 I/OSD wywołanie funkcji connect() dla gniazda z rodziny AF_INET i rodzaju SOCK_DGRAM spowoduje, że system zmieni adres, z którym gniazdo jest powiązane, na adres interfejsu używanego do osiągnięcia adresu podanego w funkcji connect(). Jeśli na przykład gniazdo typu SOCK_DGRAM zostanie powiązane z adresem INADDR_ANY, a następnie połączone z adresem a.b.c.d, system zmieni gniazdo tak, że będzie ono powiązane z adresem IP interfejsu wybranego do kierowania pakietów do adresu a.b.c.d. Ponadto, jeśli adres IP, z którym powiązane jest gniazdo, to na przykład a.b.c.e, adres a.b.c.e pojawi się podczas wywołania funkcji getsockname zamiast adresu INADDR_ANY i trzeba będzie użyć opcji SO_REUSEADDR, aby powiązać wszelkie inne gniazda dla tego samego numeru portu z adresem a.b.c.e.

Implementacja i5/OS zachowa się w tym przypadku inaczej, to jest nie zmieni lokalnego adresu z INADDR_ANY na a.b.c.e. Funkcja getsockname() będzie zwracać adres INADDR_ANY także po zrealizowaniu połączenia.

Opcje SO_SNDBUF i SO_RCVBUF

Wartości opcjom SO_SNDBUF i SO_RCVBUF w systemie BSD zapewniają większą kontrolę niż w implementacji i5/OS. W implementacji i5/OS wartości te mają znaczenie pomocnicze.

Pojęcia pokrewne

“Jak działają gniazda” na stronie 3

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

Informacje pokrewne

accept()
sendmsg()
connect()
recvfrom()
recvmsg()
bind()
getsockname()
socket()
listen()
ioctl()
getpeername()
close()

Zgodność z UNIX 98.

Opracowany przez Open Group, konsorcjum programistów i dostawców oprogramowania, standard UNIX 98 usprawnił współdziałanie systemu operacyjnego UNIX zawierając przy tym większość funkcji związanych z obsługą Internetu, dzięki którym system UNIX stał się znany.

Gniazda systemu i5/OS umożliwiają programistom pisanie aplikacji gniazd kompatybilnych ze środowiskiem UNIX 98. Obecnie IBM obsługuje dwie wersje większości funkcji API gniazd. Podstawowa funkcja API systemu i5/OS używa struktury i składni BSD (Berkeley Socket Distribution) 4.3. Druga wersja używa składni i struktur zgodnych z BSD 4.4 i specyfikacją interfejsu programistycznego UNIX 98. Definiując wartość 520 lub większą dla makra `_XOPEN_SOURCE`, można wybrać interfejs zgodny ze standardem UNIX 98.

Różnice w strukturze adresów dla aplikacji zgodnych z UNIX 98

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

Tabela 16. Porównanie struktur adresowych gniazd BSD 4.3 i UNIX 98/BSD 4.4

Struktura BSD 4.3	Struktura BSD 4.4 kompatybilna z systemem UNIX 98
Struktura adresów sockaddr	
<pre>struct sockaddr { u_short sa_family; char sa_data[14]; };</pre>	<pre>struct sockaddr { uint8_t sa_len; sa_family_t sa_family; char sa_data[14]; };</pre>
Struktura adresów sockaddr_in	
<pre>struct sockaddr_in { short sin_family; u_short sin_port; struct in_addr sin_addr; char sin_zero[8]; };</pre>	<pre>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]; };</pre>
Struktura adresów sockaddr_in6	
<pre>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; };</pre>	<pre>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; };</pre>
Struktura adresów sockaddr_un	
<pre>struct sockaddr_un { short sun_family; char sun_path[126]; };</pre>	<pre>struct sockaddr_un { uint8_t sun_len; sa_family_t sun_family; char sun_path[126]; };</pre>

Różnice w funkcjach API

Podczas kompilacji aplikacji w językach opartych na środowisku ILE z podaniem makra `_XOPEN_SOURCE` niektóre funkcje API gniazd są odwzorowywane na nazwy wewnętrzne. Nazwy te działają tak samo jak oryginalne funkcje API.

Poniższa tabela zawiera funkcje API, których to dotyczy. Pisząc aplikację gniazd w języku opartym na C można określać wewnętrzne nazwy tych funkcji API bezpośrednio. Aby przeczytać uwagi dotyczące użycia i szczegóły związane z obiema wersjami tych funkcji API, należy użyć odsyłaczy.

Tabela 17. Funkcja API i odpowiadająca jej nazwa zgodna z 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()
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()

Tabela 17. Funkcja API i odpowiadająca jej nazwa zgodna z UNIX 98 (kontynuacja)

Nazwa funkcji API	Nazwa wewnętrzna
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

“Jak działają gniazda” na stronie 3

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

Informacje pokrewne

accept()
 accept_and_recv()
 connect()
 sendmsg()
 recvfrom()
 recvmsg()
 Rbind()
 recv()
 bind()
 getsockname()
 socket()
 socketpair()
 listen()
 ioctl()
 getpeername()
 close()
 endhostent()
 endnetent()
 endprotoent()
 endservent()
 gethostbyname()
 getaddrinfo()
 gethostbyaddr()
 getnameinfo()

gethostbyaddr_r()
gethostname()
gethostent()
getnetbyaddr()
getnetbyname()
getnetent()
getprotobyname()
getprotobynumber()
getprotoent()
getsockopt()
getservbyname()
getservbyport()
getservent()
inet_addr()
inet_lnaof()
inet_makeaddr()
inet_netof()
inet_network()
send()
sendto()
sethostent()
setnetent()
setprotoent()
setservent()
setsockopt()

Przekazywanie deskryptorów pomiędzy procesami - sendmsg() i recvmsg()

Możliwość przekazania otwartego deskryptora pomiędzy zadaniami może prowadzić do nowego sposobu projektowania aplikacji klient/serwer.

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

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

- Funkcja spawn()

Uwaga: Funkcja spawn() nie jest funkcją API gniazd. Jest ona dostarczana jako część funkcji API systemu i5/OS związanych z procesami.

- Funkcje givedescriptor() i takedescriptor()
- Funkcje sendmsg() i recvmsg()

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

Funkcje `sendmsg()` i `recvmsg()` mają więcej zalet w porównaniu z funkcjami `givedescriptor()` i `takedescriptor()`:

Przenośność

Funkcje API `givedescriptor()` oraz `takedescriptor()` są niestandardowe i unikalne dla iSeries. Jeśli przenośność aplikacji pomiędzy serwerem iSeries a systemem iSeries and UNIX jest istotna, użycie funkcji API `sendmsg()` and `recvmsg()` będzie lepszym rozwiązaniem.

Komunikacja informacji sterujących

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

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

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

Wydajność

Aplikacje korzystające z funkcji `sendmsg()` i `recvmsg()` działają wydajniej w porównaniu z aplikacjami korzystającymi z funkcji `givedescriptor()` i `takedescriptor()`:

- czasu, który upłynął,
- wykorzystania jednostki centralnej,
- skalowalności.

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

Pula zadań procesów roboczych

Można ustawić pulę zadań procesów roboczych tak, aby serwer mógł przekazać deskryptor i aby otrzymało go tylko jedno 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 ID procesu roboczego

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

Adaptacyjny program serwera

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

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

Nieaktywne zadanie procesu roboczego

Funkcja `givedescriptor()` wymaga, aby zadanie procesu roboczego było aktywne, funkcja `sendmsg()` tego nie

wymaga. Zadanie wywołujące `sendmsg()` nie wymaga żadnych informacji o zadaniu procesu roboczego. Funkcja `sendmsg()` wymaga tylko ustanowienia połączenia z gniazdem `AF_UNIX`.

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

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

Jednoczesne przekazanie kilku deskryptorów

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

Odsyłacze pokrewne

“Przykład: przekazywanie deskryptorów pomiędzy procesami” na stronie 96

Przykłady funkcji `sendmsg()` i `recvmsg()` pokazują, jak zaprojektować program serwera, który korzysta z tych funkcji do obsługi połączeń przychodzących.

Informacje pokrewne

`socketpair()`

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

Opis typowej sytuacji, w której użytkownik może użyć rodziny adresów `AF_INET6`.

Sytuacja

Przypuśćmy, że użytkownik jest programistą pracującym w firmie specjalizującej się w aplikacjach gniazd dla serwerów iSeries. Aby uzyskać przewagę nad konkurencją firma zdecydowała się opracować pakiet aplikacji używających rodziny adresów `AF_INET6`, które będą mogły komunikować się z systemami o adresach IPv4 i IPv6. Zadaniem jest napisanie aplikacji przetwarzającej żądania zarówno z węzłów IPv4, jak i IPv6. Programista wie, że system i5/OS obsługuje gniazda rodziny adresów `AF_INET6` i współdziałanie z gniazdami rodziny adresów `AF_INET`. Wie także, że funkcję tę realizuje się przy użyciu formatu adresu IPv4 odwzorowanego na adres IPv6.

Cele scenariusza

Scenariusz ten ma następujące cele:

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

Wymaganie wstępne

Aby opracować aplikację realizującą te cele, wykonaj następujące czynności:

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

Szczegóły scenariusza

Poniższa ilustracja przedstawia sieć IPv6, dla której użytkownik tworzy aplikacje obsługujące żądania od klientów IPv6 i IPv4. Serwer iSeries zawiera program, który nasłuchuje i przetwarza żądania od tych klientów. Sieć składa się z dwóch oddzielnych domen, z których w jednej znajdują się wyłącznie klienci IPv4, a w drugiej - wyłącznie klienci IPv6. Nazwa domeny iSeries to iSeries.myserver.myco.com. Aplikacja serwera korzysta z rodziny adresów AF_INET6 do obsługi żądań przychodzących, w wywołaniu funkcji bind() podając parametr in6addr_any.



Odsyłacze pokrewne

“Używanie rodziny adresów AF_INET6” na stronie 27

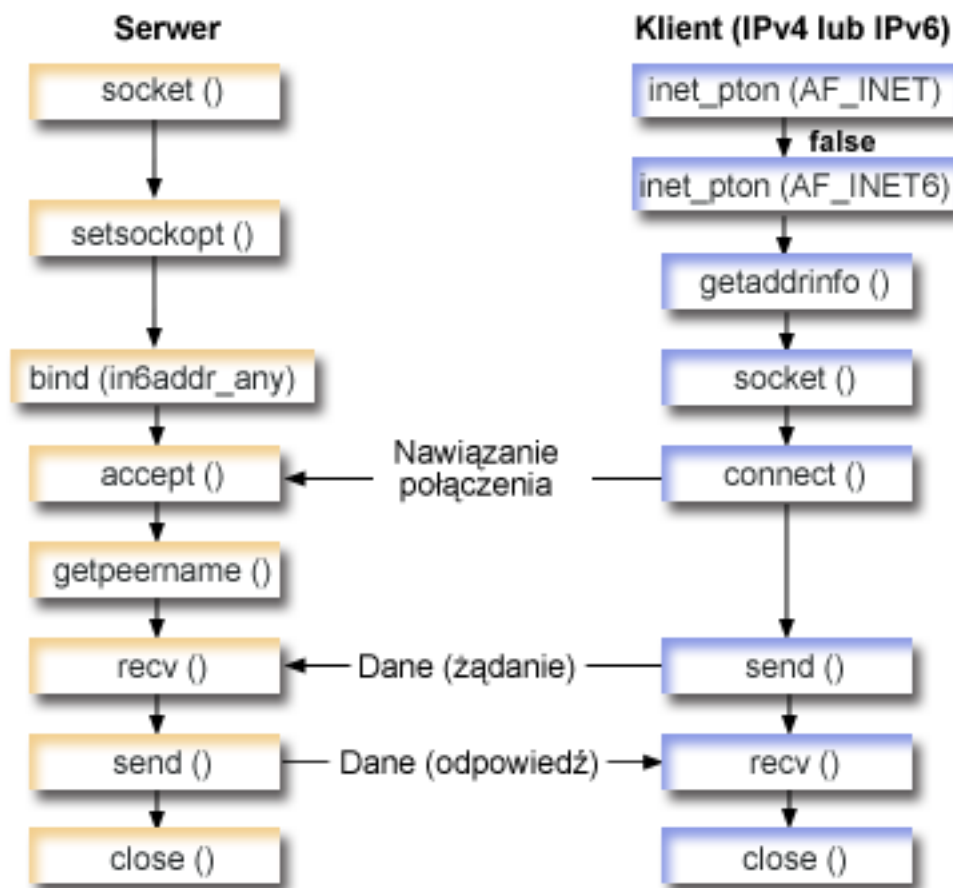
Gniazda AF_INET6 zapewniają obsługę 128-bitowych (16-bajtowych) struktur adresów protokołu IP (Internet Protocol) w wersji 6 (IPv6). Programiści mogą pisać aplikacje używające rodziny adresów AF_INET6, które będą akceptowały połączenia z klientów obsługujących zarówno protokół IPv4, jak i IPv6 lub też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

Przykład: akceptowanie połączeń od klientów IPv6 i IPv4

Zadaniem tego programu w układzie serwer/klient jest przyjmowanie żądań z sieci IPv4 (aplikacji gniazd używających rodziny adresów AF_INET) i IPv6 (aplikacji gniazd używających rodziny adresów AF_INET6).

Działające obecnie aplikacje gniazd najczęściej używają tylko rodziny adresów AF_INET, dopuszczającej protokoły TCP i UDP; jednak w miarę zwiększania się liczby adresów IPv6 sytuacja ta może ulec zmianie. Program ten może służyć do tworzenia własnych aplikacji, które będą obsługiwać obie rodziny adresów.

Poniższa ilustracja przedstawia sposób działania programu:



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

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

1. Funkcja API `socket()` określa deskryptor gniazda, który tworzy punkt końcowy. Określa również, że dla tego gniazda zostanie użyta rodzina adresów `AF_INET6`, która obsługuje IPv6 i protokół transportowy TCP (`SOCK_STREAM`).
2. Funkcja `setsockopt()` umożliwia aplikacji ponowne użycie adresu lokalnego po restarcie serwera, zanim upłynie wymagany czas oczekiwania.
3. Funkcja `bind()` dostarcza unikalnej nazwy gniazda. W tym przykładzie programista ustawia adres na `in6addr_any`, co (domyślnie) umożliwia nawiązanie połączenia z dowolnego klienta IPv4 lub IPv6, który określi port 3005 (to znaczy, że 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 `listen()` umożliwia serwerowi przyjęcie połączenia przychodzącego od klienta. W tym przykładzie programista ustawił wartość kolejki (backlog) na 10. Oznacza to, że system umieści w kolejce pierwsze 10 przychodzących żądań połączenia, a następnie odrzuci.
5. Serwer używa funkcji `accept()` do zaakceptowania żądania połączenia przychodzącego. Wywołanie funkcji `accept()` zostanie zablokowane na nieokreślony czas oczekiwania na połączenie przychodzące od klienta IPv4 lub IPv6.
6. Funkcja `getpeername()` zwraca do aplikacji adres klienta. Jeśli jest to klient IPv6, zostanie wyświetlony adres IPv4 odwzorowany na adres IPv6.

7. Funkcja `recv()` odbiera 250 bajtów danych od klienta. W tym przykładzie klient wysłał 250 bajtów danych. Na tej podstawie może użyć opcji gniazda `SO_RCVLOWAT` i określić, że funkcja `recv()` ma pozostać w uśpieniu do momentu nadejścia całych 250 bajtów danych.
8. Funkcja `send()` odsyła dane do klienta.
9. Funkcja `close()` zamyka wszystkie otwarte deskryptory gniazd.

Przebieg zdarzeń w gnieździe: żądania od klientów IPv4 lub IPv6

Uwaga: Poniższego przykładu klienta można użyć z innymi projektami aplikacji serwera, które akceptują żądania z węzłów IPv4 i IPv6. Z tym przykładem klienta mogą współpracować inne typy serwerów.

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

Poniższy kod przykładowy przedstawia aplikację serwera w tym scenariuszu.

Uwaga: Korzystając z przykładowego kodu, użytkownik akceptuje warunki określone w temacie “Informacje dotyczące licencji na kod” na stronie 192.

```

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

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

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

    /*****
    /* Pętla do/while(FALSE) ułatwia realizację procedur czyszczących w */
    /* przypadku błędu. Funkcja close() dla poszczególnych deskryptorów */

```

```

/* gniazd jest uruchamiana jednokrotnie na samym końcu programu. */
/*****
do
{

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

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

    /*****/
    /* Po utworzeniu deskryptora gniazda funkcja bind() pobiera */
    /* unikalną nazwę gniazda. W tym przykładzie użytkownik ustawia */
    /* adres na in6addr_any, który (domyślnie) umożliwia nawiązywanie */
    /* połączeń z dowolnego klienta IPv4 lub IPv6, który określi port */
    /* 3005. (To znaczy, że funkcja bind jest uruchamiana dla IPv4 */
    /* i IPv6. Zachowanie to można w razie potrzeby zmodyfikować */
    /* za pomocą opcji gniazda IPV6_V6ONLY na poziomie IPPROTO_IPV6. */
    /*****/
    memset(&serveraddr, 0, sizeof(serveraddr));
    serveraddr.sin6_family = AF_INET6;
    serveraddr.sin6_port = htons(SERVER_PORT);
    /*****/
    /* Uwaga: Aplikacje używają struktury in6addr_any w podobny sposób, */
    /* jak makra INADDR_ANY w IPv4. Stała symboliczna IN6ADDR_ANY_INIT */
    /* również istnieje, ale może zostać użyta tylko do zainicjowania */
    /* struktury in^_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 bind()");
        break;
    }

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

```

```

/*****/
if (listen(sd, 10) < 0)
{
    perror("Niepowodzenie listen()");
    break;
}

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

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

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

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

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

/*****/

```

```

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

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

} while (FALSE);

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

```

Odsyłacze pokrewne

“Przykłady: projekty aplikacji zorientowanych na połączenie” na stronie 86

W systemie iSeries istnieje wiele możliwości realizacji zorientowanego na połączenie serwera gniazd. Do utworzenia własnych programów zorientowanych na połączenie można użyć poniższych programów przykładowych.

“Przykład: klient IPv4 lub IPv6”

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

Informacje pokrewne

```

socket()
setsockopt()
bind()
listen()
accept()
getpeername()
recv()
send()
close()
inet_pton()
getaddrinfo()
connect()

```

Przykład: klient IPv4 lub IPv6

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

Uwaga: Korzystając z przykładowego kodu, użytkownik akceptuje warunki określone w temacie “Informacje dotyczące licencji na kod” na stronie 192.

```

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

/*****/

```

```

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

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

/* Przekaż 1 parametr, który albo jest */
/* adresem, albo nazwą hosta serwera lub */
/* ustaw nazwę serwera zmienną #define */
/* SERVER_NAME. */
void main(int argc, char *argv[])
{
    /*****
    /* Definicje zmiennych i struktur. */
    /*****
    int sd=-1, rc, bytesReceived=0;
    char buffer[BUFFER_LENGTH];
    char server[NETDB_MAX_HOST_NAME_LENGTH];
    char servport[] = "3005";
    struct in6_addr serveraddr;
    struct addrinfo hints, *res=NULL;

    /*****
    /* Pętla do/while(FALSE) ułatwia realizację procedur czyszczących w */
    /* przypadku błędu. Funkcja close() dla deskryptora gniazda jest */
    /* wykonywana tylko raz na końcu programu i opróżnia listę adresów. */
    /*****
    do
    {
        /*****
        /* Jeśli został przekazany argument, należy go użyć jako nazwy */
        /* serwera, w przeciwnym razie należy użyć zmiennej określonej */
        /* w makrze #define znajdującym się na początku programu. */
        /*****
        if (argc > 1)
            strcpy(server, argv[1]);
        else
            strcpy(server, SERVER_NAME);

        memset(&hints, 0x00, sizeof(hints));
        hints.ai_flags = AI_NUMERICSERV;
        hints.ai_family = AF_UNSPEC;
        hints.ai_socktype = SOCK_STREAM;
        /*****
        /* Sprawdza, czy został dostarczony adres serwera za pomocą funkcji */
        /* inet_pton(), która przekształca tekstową postać adresu do */
        /* binarnej. Jeśli adres jest liczbowy, funkcja getaddrinfo() nie */
        /* powinna wykonywać rozstrzygnięcia nazwy. */
        /*****
        rc = inet_pton(AF_INET, server, &serveraddr);
        if (rc == 1) /* valid IPv4 text address? */
        {
            hints.ai_family = AF_INET;
            hints.ai_flags |= AI_NUMERICHOST;
        }
        else

```

```

{
    rc = inet_pton(AF_INET6, server, &serveraddr);
    if (rc == 1) /* poprawny adres tekstowy IPv6? */

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

/*****/
/* Funkcja socket() zwraca deskryptor gniazda stanowiącego punkt
/* końcowy. Instrukcja ta identyfikuje również rodzinę adresów,
/* typ gniazda i protokół na podstawie informacji zwróconych przez
/* getaddrinfo().
/*****/
sd = socket(res->ai_family, res->ai_socktype, res->ai_protocol);
if (sd < 0)
{
perror("Niepowodzenie socket()");
    break;
}

/*****/
/* Aby nawiązać połączenie z serwerem, zostanie użyta funkcja
/* connect().
/*****/
rc = connect(sd, res->ai_addr, res->ai_addrlen);
if (rc < 0)
{
    /*****/
    /* Uwaga: res jest dowiązaną listą adresów znalezionych dla
    /* serwera. Jeśli funkcja connect() nie powiedzie się przy
    /* pierwszym, mogą zostać wypróbowane kolejne adresy na liście.
    /*****/
    perror("Niepowodzenie connect()");
    break;
}

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

/*****/
/* W tym przykładzie wiadomo, że serwer odpowie wysłaniem tych
/* samych 250 bajtów, które 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 przysyłanych w oddzielnych pakietach, */
/* funkcja recv() będzie wielokrotnie uruchamiana, dopóki */
/* wszystkie 250 bajtów nie zostanie odebrane. */
/*****/
while (bytesReceived < BUFFER_LENGTH)
{
    rc = recv(sd, & buffer[bytesReceived],
              BUFFER_LENGTH - bytesReceived, 0);
    if (rc < 0)
    {
        perror("Niepowodzenie recv()");
        break;
    }
    else if (rc == 0)
    {
        printf("Serwer zamknął połączenie\n");
        break;
    }
    }

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

} while (FALSE);

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

```

Odsyłacze pokrewne

“Przykład: akceptowanie połączeń od klientów IPv6 i IPv4” na stronie 75

Zadaniem tego programu w układzie serwer/klient jest przyjmowanie żądań z sieci IPv4 (aplikacji gniazd używających rodziny adresów AF_INET) i IPv6 (aplikacji gniazd używających rodziny adresów AF_INET6).

“Przykład: ogólny program klienta” na stronie 108

Poniższy kod zawiera przykłady podstawowych zadań klienta.

Zalecenia dotyczące projektowania aplikacji używających gniazd

Aby projektować aplikacje używające gniazd, należy określić wymagania funkcjonalne, cele i potrzeby tej aplikacji.

Należy również rozważyć wymagania dotyczące wydajności oraz wpływ zasobów systemowych na aplikację. Poniższa lista zaleceń może pomóc rozwiązać te kwestie i wskazać efektywniejsze metody wykorzystania gniazd i projektowania aplikacji używających gniazd:

Tabela 18. 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.
Używając asynchronicznych operacji we/wy, dostosuj liczbę wątków w procesie do liczby optymalnej dla liczby klientów, którzy mają być przetwarzani.	W przypadku zbyt małej liczby wątków niektórzy klienci mogą przekroczyć czas oczekiwania na obsługę. W przypadku zbyt dużej liczby wątków niektóre zasoby systemowe nie są wykorzystywane efektywnie. Uwaga: Lepiej, gdy wątków jest za dużo niż za mało.	Aplikacje używające gniazd, korzystające z asynchronicznych operacji we/wy.
Projektuj aplikacje używające gniazd tak, aby uniknąć używania postflag dla wszystkich operacji rozpoczynających asynchroniczne operacje we/wy.	Pozwala to uniknąć dodatkowego obciążenia związanego z przejściem do portu zakończenia, jeśli operacja została wykonana synchronicznie.	Aplikacje używające gniazd, korzystające z asynchronicznych operacji we/wy.
Korzystaj z funkcji send() i recv() zamiast read() i write().	Funkcje send() i recv() mają nieco poprawioną wydajność i możliwości obsługi serwisowej w stosunku do funkcji read() i write().	Dowolny program używający gniazd, który rozróżnia używanie deskryptorów gniazd od deskryptorów plików.
Używaj dla gniazda opcji odbierania SO_RCVLOWAT, aby uniknąć zapętlenia podczas operacji odbierania, dopóki nie zostaną przysłane wszystkie dane.	Dzięki temu aplikacja może oczekiwać na minimalną ilość danych do odebrania przez gniazdo przed spełnieniem warunku dla zablokowanej operacji odbioru.	Dowolna aplikacja używająca gniazd, która odbiera dane.
Używaj opcji MSG_WAITALL, aby uniknąć zapętlenia podczas operacji odbierania, dopóki nie zostaną przysłane wszystkie dane.	Dzięki temu aplikacja może oczekiwać na całą zawartość buforu operacji odbierania, zanim zostanie spełniony warunek zablokowanej operacji odbioru.	Dowolna aplikacja używająca gniazd, która odbiera dane i wie z góry, ile danych ma oczekiwać.
Używaj funkcji sendmsg() i recvmsg() zamiast funkcji givedescriptor() i takedescriptor().	Zalety tego rozwiązania zostały szczegółowo opisane w sekcji "Przekazywanie deskryptorów pomiędzy procesami - sendmsg() i recvmsg()" na stronie 72.	Dowolna aplikacja używająca gniazd, która przekazuje deskryptory gniazd lub plików pomiędzy procesami.
Używając funkcji select(), próbuj unikać dużej liczby deskryptorów w zestawie operacji odczytu, zapisu lub w zestawie wyjątków. Uwaga: W przypadku dużej liczby deskryptorów używanych z przetwarzaniem funkcji select(), należy kierować się powyższym zaleceniem dotyczącym używania asynchronicznych operacji we/wy.	W przypadku dużej liczby deskryptorów dla zestawu operacji odczytu, zapisu lub w zestawie wyjątków, przy każdym wywołaniu funkcji select() wykonywanych jest wiele powtarzających się czynności. Po wykonaniu funkcji select() należy wykonać 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 aktywnych deskryptorów dla funkcji select().
Zapisz zestawy operacji odczytu zapisu lub zestawy wyjątków przed wywołaniem funkcji select(), aby uniknąć odbudowywania zestawów podczas każdego wywołania funkcji select().	Ta czynność eliminuje dodatkowe czynności związane z przebudowywaniem zestawów operacji odczytu, zapisu i zestawów wyjątków za każdym razem, gdy planowane jest wywołanie funkcji select().	Dowolna aplikacja używająca gniazd, w której wykorzystuje się funkcję select() z dużą liczbą deskryptorów aktywnych dla przetwarzania odczytu, zapisu lub wyjątków.

Tabela 18. Projektowanie aplikacji używających gniazd (kontynuacja)

Zalecenie	Przyczyna	Najlepsze zastosowanie
Nie używaj funkcji select() jako zegara. Zamiast niej używaj funkcji sleep(). Uwaga: Jeśli granulacja funkcji sleep() nie jest odpowiednia, może być konieczne użycie funkcji select() jako zegara. W takim przypadku ustaw maksymalną liczbę deskryptorów na 0, a zestawy operacji odczytu, zapisu i zestawy wyjątków na NULL.	Zapewni to lepsze odpowiedzi zegara i zmniejszy obciążenie systemu.	Dowolna aplikacja używająca gniazd, w której funkcja select() jest stosowana wyłącznie jako zegar.
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 używana jest funkcja select(), 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, gdzie używa się funkcji DosSetRelMaxFH() i select().
Ustaw wszystkie deskryptory gniazd w zestawach operacji odczytu lub zapisu jako nieblokujące. Kiedy deskryptor staje się dostępny do odczytu lub zapisu, zapętla się i przyjmuje lub wysyła wszystkie dane dopóki nie zostanie zwrócona wartość EWOULDBLOCK.	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 stosowana jest funkcja select().
Wykorzystując przetwarzanie funkcji select(), określ tylko niezbędne zestawy.	Większość aplikacji nie potrzebuje określania zestawu wyjątków ani zestawu operacji zapisu.	Dowolna aplikacja używająca gniazd, w której stosowana jest funkcja select().
Używaj funkcji API GSKit zamiast funkcji API SSL.	Zarówno funkcja Global Secure Toolkit (GSKit) jak i funkcje SSL APIi5/OS umożliwiają tworzenie chronionych aplikacji dla rodziny adresów AF_INET albo AF_INET6, używających gniazd SOCK_STREAM. Ponieważ funkcje API GSKit są obsługiwane przez wszystkie systemy IBM, są one preferowanymi funkcjami API dla tworzenia chronionych aplikacji. Funkcje API SSL_APIs występują tylko w systemie operacyjnym i5/OS.	Każda aplikacja używająca gniazd, która wymaga obsługi przetwarzania SSL/TLS.
Nie używaj sygnałów.	Obciążenie związane z sygnałami jest kosztowne (na wszystkich platformach, nie tylko dla iSeries). Lepiej zaprojektować aplikację, która będzie używać asynchronicznych operacji we/wy lub funkcji select().	Każdy programista, który zastanawia się nad wykorzystaniem sygnałów w swojej aplikacji używającej gniazd.
W miarę możliwości używaj procedur niezależnych od protokołu, takich jak inet_ntop(), inet_pton(), getaddrinfo() i getnameinfo().	Warto używać tych funkcji API (zamiast inet_ntoa(), inet_addr(), gethostbyname() i gethostbyaddr()) nawet wtedy, gdy aplikacja nie ma obsługiwać IPv6, aby ułatwić późniejszą migrację.	Dowolna aplikacja AF_INET lub AF_INET6 korzystająca z procedur sieciowych.

Tabela 18. Projektowanie aplikacji używających gniazd (kontynuacja)

Zalecenie	Przyczyna	Najlepsze zastosowanie
Użyj <code>sockaddr_storage</code> do zadeklarowania obszaru pamięci dla dowolnego adresu z tej rodziny adresów.	Upraszcza pisanie kodu przenośnego pomiędzy wieloma rodzinami adresów i platformami. Deklaruje wystarczającą ilość pamięci, aby pomieściła największą rodzinę adresów i zapewnia poprawne wyrównanie adresu.	Dowolna aplikacja używająca gniazd, która przechowuje adresy.

Pojęcia pokrewne

“Asynchroniczne operacje we/wy” na stronie 43

Funkcje API asynchronicznych operacji we/wy udostępniają metodę dla wątków modelu klient serwer, umożliwiającą realizację współbieżnych operacji we/wy z efektywnym wykorzystaniem pamięci.

Odsyłacze pokrewne

“Przykład: korzystanie z asynchronicznych operacji we/wy” na stronie 111

Aplikacja tworzy port we/wy zakończenia za pomocą funkcji `API QsoCreateIOCompletionPort()`.

“Przykład: nieblokujące operacje we/wy i funkcja `select()`” na stronie 151

Poniższy program przykładowy wykorzystuje nieblokujące operacje we/wy i funkcję `API select()`.

“Przykład: przekazywanie deskryptorów pomiędzy procesami” na stronie 96

Przykłady funkcji `sendmsg()` i `recvmsg()` pokazują, jak zaprojektować program serwera, który korzysta z tych funkcji do obsługi połączeń przychodzących.

Informacje pokrewne

`DosSetRelMaxFH()`

Przykłady: projekty aplikacji używających gniazd

Poniższe przykłady zawierają wiele programów ilustrujących zaawansowane koncepcje dotyczące gniazd. Można ich użyć do tworzenia własnych aplikacji, które będą realizowały podobne zadania.

Przykładom towarzyszą ilustracje i lista wywołań, które ilustrują przebieg zdarzeń w poszczególnych aplikacjach. Można użyć narzędzia `Xsockets` do interaktywnego wypróbowania niektórych funkcji API w tych programach lub dokonać odpowiednich zmian, aby przystosować je do środowiska, w którym będą pracować.

Przykłady: projekty aplikacji zorientowanych na połączenie

W systemie `iSeries` istnieje wiele możliwości realizacji zorientowanego na połączenie serwera gniazd. Do utworzenia własnych programów zorientowanych na połączenie można użyć poniższych programów przykładowych.

Mimo iż możliwe są inne projekty serwerów używających gniazd, poniższe przykłady są używane najczęściej.

Serwer iteracyjny

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

Serwer współbieżny

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

W przypadku wielu współbieżnych klientów w sieci zaleca się użycie funkcji API gniazd obsługujących asynchroniczne operacje we/wy. Funkcje te zapewniają największą wydajność w sieciach z wieloma klientami działającymi współbieżnie.

- Serwer `spawn()` i proces roboczy `spawn()`

Funkcja API `spawn()` używana jest do utworzenia zadania, które obsłuży każde żądanie przychodzące. Po zakończeniu działania funkcji `spawn()` serwer może czekać przy funkcji API `accept()` na odebranie następnego połączenia przychodzącego.

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

- Serwer `spawn()` i proces roboczy `spawn()`

Funkcje `sendmsg()` i `recvmsg()` są używane do obsługi połączeń przychodzących. Przy pierwszym uruchomieniu zadania serwera prestartuje ono wszystkie zadania procesu roboczego.

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

W poprzednich funkcjach API, proces roboczy nie wykonuje żadnych czynności, dopóki serwer nie odebrał żądania połączenia przychodzącego. Korzystając z wielu funkcji API `accept()`, możliwa jest zamiana każdego zadania procesu roboczego 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 daje każdemu z nich gniazdo do nasłuchu. Następnie wszystkie zadania procesów roboczych wywołują funkcję API `accept()`. Gdy klient próbuje połączyć się z serwerem, jedno wywołanie funkcji `accept()` kończy działanie i ten proces roboczy obsługuje połączenie.

Pojęcia pokrewne

“Asynchroniczne operacje we/wy” na stronie 43

Funkcje API asynchronicznych operacji we/wy udostępniają metodę dla wątków modelu klient serwer, umożliwiającą realizację współbieżnych operacji we/wy z efektywnym wykorzystaniem pamięci.

Odsyłacze pokrewne

“Przykład: akceptowanie połączeń od klientów IPv6 i IPv4” na stronie 75

Zadaniem tego programu w układzie serwer/klient jest przyjmowanie żądań z sieci IPv4 (aplikacji gniazd używających rodziny adresów `AF_INET`) i IPv6 (aplikacji gniazd używających rodziny adresów `AF_INET6`).

“Przykład: korzystanie z asynchronicznych operacji we/wy” na stronie 111

Aplikacja tworzy port we/wy zakończenia za pomocą funkcji API `QsoCreateIOCompletionPort()`.

“Przykład: ogólny program klienta” na stronie 108

Poniższy kod zawiera przykłady podstawowych zadań klienta.

Informacje pokrewne

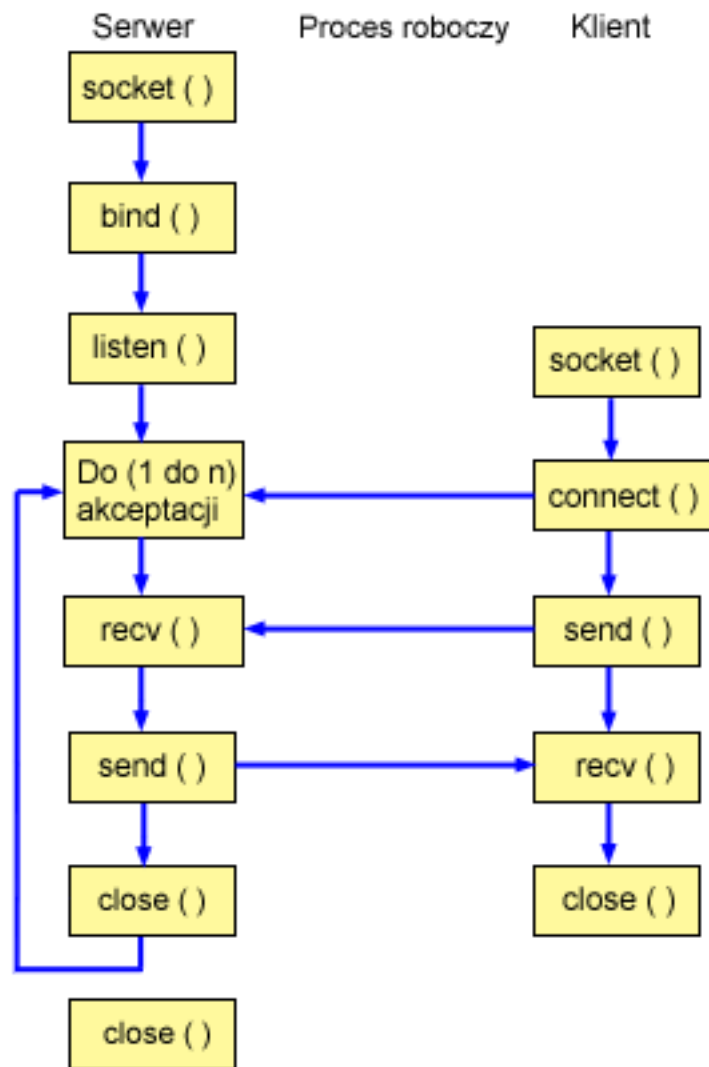
`accept()`

`spawn()`

Przykład: pisanie programu serwera iteracyjnego

Poniżej znajduje się przykład pojedynczego zadania serwera, które obsługuje wszystkie połączenia przychodzące. Po zakończeniu działania funkcji API `accept()` serwer obsługuje całą transakcję.

Rysunek przedstawia, jak zadania serwera i klienta współdziałają ze sobą, gdy w systemie używany jest program serwera iteracyjnego.



Przebieg zdarzeń w gnieździe: serwer iteracyjny

Poniżej wyjaśniono sekwencję wywołań funkcji gniazd, przedstawionych na powyższej ilustracji. Opisano także relacje pomiędzy aplikacją serwera a aplikacją procesu roboczego. Każdy zbiór wywołań zawiera odsyłacze do uwag dotyczących użycia poszczególnych funkcji API. Aby uzyskać szczegółowe informacje dotyczące użycia tych funkcji API, należy użyć poniższych odsyłaczy. Poniżej opisano kolejność wywołań funkcji w przykładowej aplikacji serwera iteracyjnego:

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

Uwaga: Korzystając z przykładowego kodu, użytkownik akceptuje warunki określone w temacie “Informacje dotyczące licencji na kod” na stronie 192.

```
/******  
/* Aplikacja serwera iteracyjnego */  
/******  
#include <stdio.h>  
#include <stdlib.h>  
#include <sys/socket.h>  
#include <netinet/in.h>  
  
#define SERVER_PORT 12345  
  
main (int argc, char *argv[])  
{  
    int    i, len, num, rc, on = 1;  
    int    listen_sd, accept_sd;  
    char   buffer[80];  
    struct sockaddr_in  addr;  
  
    /******  
    /* Jeśli podano argument, użyj go do sterowania */  
    /* liczbą połączeń przychodzących */  
    /******  
    if (argc >= 2)  
        num = atoi(argv[1]);  
    else  
        num = 1;  
  
    /******  
    /* Tworzy gniazdo strumienia AF_INET do */  
    /* odbierania połączeń przychodzących */  
    /******  
    listen_sd = socket(AF_INET, SOCK_STREAM, 0);  
    if (listen_sd < 0)  
    {  
        perror("Niepowodzenie socket()");  
        exit(-1);  
    }  
  
    /******  
    /* Umożliwia ponowne użycie deskryptora gniazda */  
    /******  
    rc = setsockopt(listen_sd,  
                    SOL_SOCKET, SO_REUSEADDR,  
                    (char *)&on, sizeof(on));  
  
    if (rc < 0)  
    {  
        perror("Niepowodzenie setsockopt()");  
        close(listen_sd);  
        exit(-1);  
    }  
  
    /******  
    /* Powiąż gniazdo */  
    /******  
    memset(&addr, 0, sizeof(addr));  
    addr.sin_family = AF_INET;  
    addr.sin_addr.s_addr = htonl(INADDR_ANY);  
    addr.sin_port = htons(SERVER_PORT);  
    rc = bind(listen_sd,  
              (struct sockaddr *)&addr, sizeof(addr));  
    if (rc < 0)  
    {  
        perror("Niepowodzenie bind()");  
        close(listen_sd);  
        exit(-1);  
    }  
}
```

```

}

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

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

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

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

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

    /*****/
}

```



```

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

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

```

Odsyłacze pokrewne

“Przykład: ogólny program klienta” na stronie 108

Poniższy kod zawiera przykłady podstawowych zadań klienta.

Informacje pokrewne

recv()
bind()
socket()
listen()
accept()
send()
close()

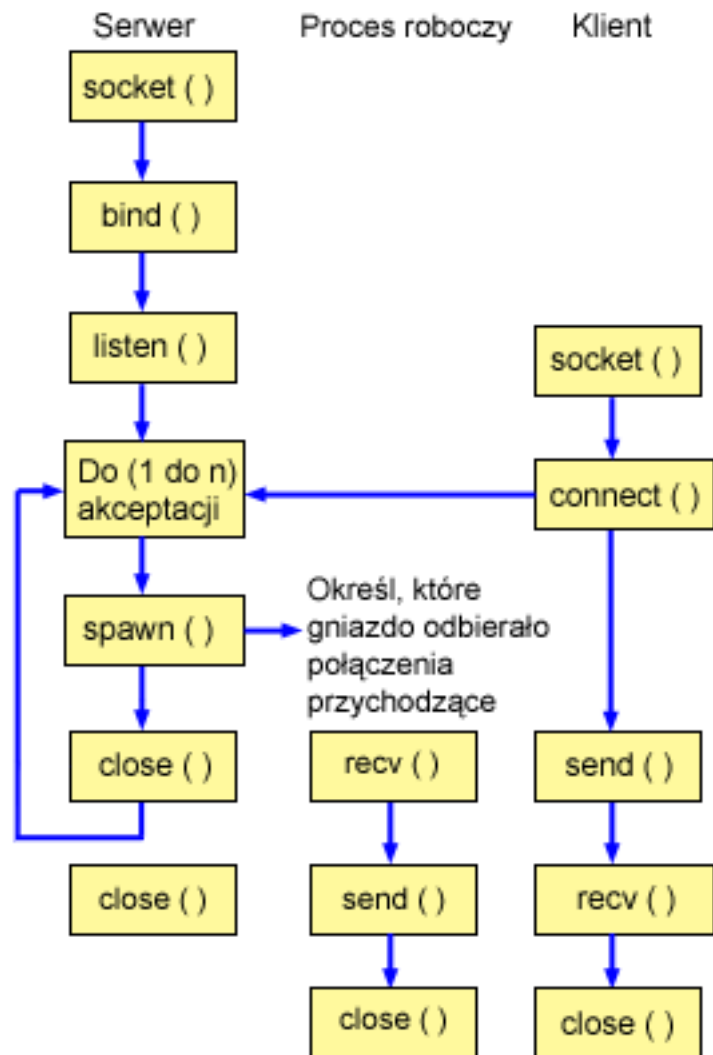
Przykład: używanie funkcji API spawn() do tworzenia procesów potomnych

Niniejszy przykład pokazuje, jak program serwera może użyć funkcji API spawn() do utworzenia procesu potomnego, który dziedziczy deskryptor gniazda po procesie macierzystym.

Zadanie serwera czeka na połączenie przychodzące, a następnie wywołuje funkcję spawn() w celu utworzenia procesów potomnych do obsługi połączenia. W ramach wywołania funkcji spawn() proces potomny dziedziczy następujące atrybuty:

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

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



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

Na poniższej ilustracji wyjaśniono sekwencję wywołań funkcji gniazd. Opisano także relacje pomiędzy aplikacją serwera a przykładowym procesem roboczym. Każdy zbiór wywołań zawiera odwołania do uwag dotyczących użycia poszczególnych funkcji API. Aby uzyskać szczegółowe informacje dotyczące użycia tych funkcji API, należy użyć poniższych odsyłaczy. Pierwszy przykład wykorzystuje następujące wywołania funkcji gniazd dla tworzenia procesu potomnego, za pomocą funkcji spawn():

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

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

Przebieg zdarzeń w gnieździe: zadanie procesu roboczego utworzone przy użyciu funkcji spawn()

Kolejny przykład wykorzystuje następującą sekwencję wywołań funkcji:

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

Informacje pokrewne

spawn()
bind()
socket()
listen()
accept()
close()
send()
recv()

Przykład: tworzenie serwera używającego funkcji spawn():

Przykład ten ilustruje, jak używać funkcji API spawn() do tworzenia procesu potomnego, który dziedziczy deskryptor gniazda od procesu macierzystego.

Uwaga: Korzystając z przykładowego kodu, użytkownik akceptuje warunki określone w temacie “Informacje dotyczące licencji na kod” na stronie 192.

```
/******  
/* Aplikacja tworzy za pomocą funkcji spawn() proces potomny. */  
/******  
  
#include <stdio.h>  
#include <stdlib.h>  
#include <sys/socket.h>  
#include <netinet/in.h>  
#include <spawn.h>  
  
#define SERVER_PORT 12345  
  
main (int argc, char *argv[])  
{  
    int    i, num, pid, rc, on = 1;  
    int    listen_sd, accept_sd;  
    int    spawn_fds[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;
```

```

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

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

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

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

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

/*****/
/* Przejdź przez pętlę raz dla każdego połączenia*/
/*****/
for (i=0; i < num; i++)
{
    /*****/
    /* Oczekiwanie na połączenie przychodzące */
    /*****/
    printf("Iteracja: %d\n", i+1);
    printf(" oczekuje na accept()\n");
}

```

```

accept_sd = accept(listen_sd, NULL, NULL);
if (accept_sd < 0)
{
    perror("Niepowodzenie accept()");
    close(listen_sd);
    exit(-1);
}
printf(" accept zakończone powodzeniem\n");

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

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

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

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

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

```

Odsyłacze pokrewne

“Przykład: umożliwianie przekazania buforu danych do zadania procesu roboczego”

Przykład ten zawiera kod umożliwiający przekazanie buforu danych z zadania klienta do zadania procesu roboczego i odesłanie go z powrotem.

Przykład: umożliwianie przekazania buforu danych do zadania procesu roboczego:

Przykład ten zawiera kod umożliwiający przekazanie buforu danych z zadania klienta do zadania procesu roboczego i odesłanie go z powrotem.

Uwaga: Korzystając z przykładowego kodu, użytkownik akceptuje warunki określone w temacie “Informacje dotyczące licencji na kod” na stronie 192.

```

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

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

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

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

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

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

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

```

Odsyłacze pokrewne

“Przykład: tworzenie serwera używającego funkcji `spawn()`” na stronie 93

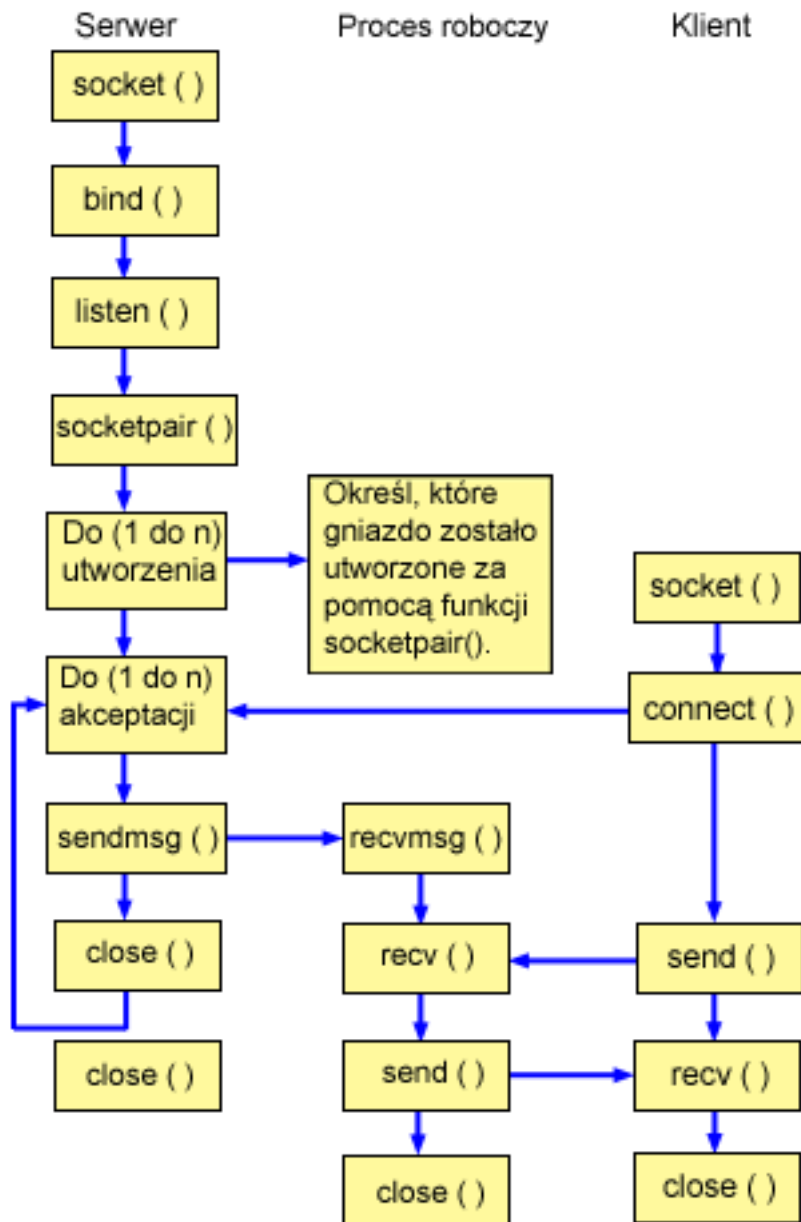
Przykład ten ilustruje, jak używać funkcji API `spawn()` do tworzenia procesu potomnego, który dziedziczy deskryptor gniazda od procesu macierzystego.

Przykład: przekazywanie deskryptorów pomiędzy procesami

Przykłady funkcji `sendmsg()` i `recvmsg()` pokazują, jak zaprojektować program serwera, który korzysta z tych funkcji do obsługi połączeń przychodzących.

Gdy serwer jest uruchamiany, tworzy pulę zadań procesów roboczych. Te uprzednio przypisane (utworzone) zadania procesów roboczych oczekują na moment, w którym będą potrzebne. Gdy zadanie klienta łączy się z serwerem, serwer przypisuje połączenie przychodzące do jednego z zadań procesów roboczych.

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



Przebieg zdarzeń w gnieździe: serwer używający funkcji `sendmsg()`

Na poniższej ilustracji wyjaśniono sekwencję wywołań funkcji gniazd. Opisano także relacje pomiędzy aplikacją serwera a przykładowym procesem roboczym. Pierwszy przykład wykorzystuje następujące wywołania funkcji gniazd dla tworzenia procesu potomnego, za pomocą funkcji `sendmsg()` i funkcji `recvmsg()`:

1. Funkcja `socket()` zwraca deskryptor gniazda odpowiadający punktowi końcowemu. Instrukcja ta informuje również, że dla tego gniazda zostanie użyta rodzina adresów INET (Internet Protocol) z transportem TCP transport (`SOCK_STREAM`).

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

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

Kolejny przykład wykorzystuje następującą sekwencję wywołań funkcji:

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

Odsyłacze pokrewne

“Przekazywanie deskryptorów pomiędzy procesami - `sendmsg()` i `recvmsg()`” na stronie 72

Możliwość przekazania otwartego deskryptora pomiędzy zadaniami może prowadzić do nowego sposobu projektowania aplikacji klient/serwer.

“Zalecenia dotyczące projektowania aplikacji używających gniazd” na stronie 83

Aby projektować aplikacje używające gniazd, należy określić wymagania funkcjonalne, cele i potrzeby tej aplikacji.

“Przykład: ogólny program klienta” na stronie 108

Poniższy kod zawiera przykłady podstawowych zadań klienta.

Informacje pokrewne

`spawn()`

`bind()`

`socket()`

`listen()`

`accept()`

`close()`

`socketpair()`

`sendmsg()`

`recvmsg()`

`send()`

`recv()`

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

Przykład ten ilustruje, jak używać funkcji API `sendmsg()` do tworzenia puli zadań procesów roboczych.

Uwaga: Korzystając z przykładowego kodu, użytkownik akceptuje warunki określone w temacie “Informacje dotyczące licencji na kod” na stronie 192.

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

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

    /* **** */
    /* Umożliwia ponowne użycie deskryptora gniazda */
    /* **** */
    rc = setsockopt(listen_sd,
                    SOL_SOCKET, SO_REUSEADDR,
                    (char *)&on, sizeof(on));

    if (rc < 0)
    {
        perror("Niepowodzenie setsockopt()");
        close(listen_sd);
        exit(-1);
    }

    /* **** */
    /* Powiąż gniazdo */
    /* **** */
    memset(&addr, 0, sizeof(addr));
    addr.sin_family = AF_INET;
    addr.sin_addr.s_addr = htonl(INADDR_ANY);
    addr.sin_port = htons(SERVER_PORT);
    rc = bind(listen_sd,
```

```

        (struct sockaddr *)&addr, sizeof(addr));
if (rc < 0)
{
    perror("Niepowodzenie bind()");
    close(listen_sd);
    exit(-1);
}

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

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

/*****
/* Inicjowanie parametrów przed wejściem w pętlę */
/*                                          */
/* Deskryptor gniazda proc. rob. jest odwzoro-  */
/* wywany na deskryptor 0 programu potomnego.  */
*****/
memset(&inherit, 0, sizeof(inherit));
spawn_argv[0] = NULL;
spawn_envp[0] = NULL;
spawn_fdmmap[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_fdmmap, &inherit,
                spawn_argv, spawn_envp);
    if (pid < 0)
    {
        perror("Niepowodzenie spawn()");
        close(listen_sd);
        close(server_sd);
        close(worker_sd);
        exit(-1);
    }
    printf(" Proces roboczy = %d\n", pid);
}

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

```

```

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

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

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

/*****/
/* Żadne dane nie są wysyłane, więc nie po- */
/* trzeba podawać wartości dla żadnego z pól */
/* msg_iov. */
/* Wartość memset struktury nagłówka komunika-*/
/* tu ustawi wskaźnik msg_iov na NULL */
/* i w polu msg_iovcnt wartość 0. */
/*****/

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

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

```

```

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

/*****
/* Zamknięcie gniazda serwera i nasłuchiwanie */
/*****
close(server_sd);
close(listen_sd);
}

```

Odsyłacze pokrewne

“Przykład: ogólny program klienta” na stronie 108

Poniższy kod zawiera przykłady podstawowych zadań klienta.

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

Przykład ten ilustruje, jak używać zadania klienta funkcji API recvmsg() do odbierania zadań procesów roboczych.

Uwaga: Korzystając z przykładowego kodu, użytkownik akceptuje warunki określone w temacie “Informacje dotyczące licencji na kod” na stronie 192.

```

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

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

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

    /*****
    /* Wypełnienie pola msg_accrights, aby można

```

```

/* było odebrać deskryptor */
/*****
msg.msg_accrights = (char *)&pass_sd;
msg.msg_accrightslen = sizeof(pass_sd);

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

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

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

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

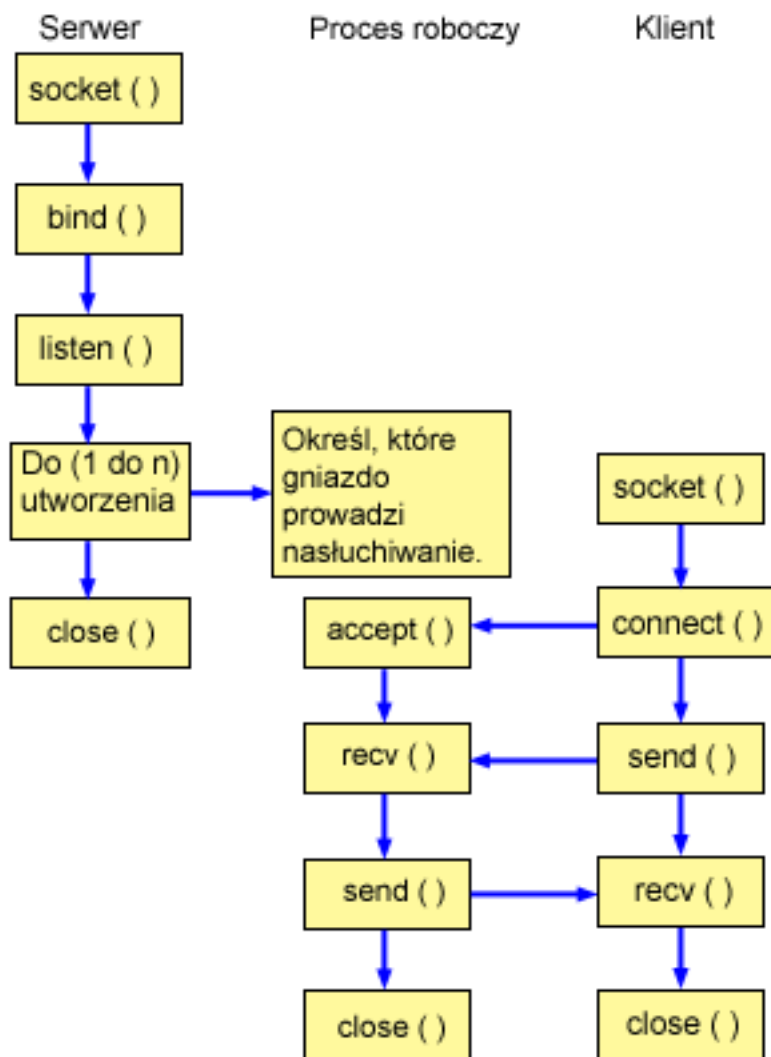
```

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

Przykłady te ilustrują sposób projektowania programu serwera, który używa modelu wielu funkcji accept() do obsługi żądań połączeń przychodzących.

Gdy uruchamiany jest serwer wielu funkcji `accept()`, uruchamia on funkcje `socket()`, `bind()` i `listen()`. Następnie tworzy pulę zadań procesów roboczych i przypisuje do każdego zadania gniazdo nasłuchujące. Następnie każdy proces roboczy wielu funkcji `accept()` wywołuje funkcję `accept()`.

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



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

Na poniższej ilustracji wyjaśniono sekwencję wywołań funkcji gniazd. Opisano także relacje pomiędzy aplikacją serwera a przykładowym procesem roboczym. Każdy zbiór wywołań zawiera odsyłacze do uwag dotyczących użycia poszczególnych funkcji API. Aby uzyskać szczegółowe informacje dotyczące użycia tych funkcji API, należy użyć poniższych odsyłaczy. Pierwszy przykład wykorzystuje następujące wywołania funkcji gniazd dla tworzenia procesu potomnego:

1. Funkcja `socket()` zwraca deskryptor gniazda odpowiadający punktowi końcowemu. Instrukcja ta informuje również, że dla tego gniazda zostanie użyta rodzina adresów INET (Internet Protocol) z transportem TCP transport (`SOCK_STREAM`).

2. Po utworzeniu deskryptora gniazda funkcja bind() pobiera unikalną nazwę gniazda.
3. Funkcja listen() umożliwia serwerowi przyjęcie połączenia przychodzącego od klienta.
4. Funkcja spawn() tworzy poszczególne zadania procesu roboczego.
5. W tym przykładzie pierwsze wywołanie funkcji close() zamyka gniazdo nasłuchujące.

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

Kolejny przykład wykorzystuje następującą sekwencję wywołań funkcji:

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

Odsyłacze pokrewne

“Przykład: ogólny program klienta” na stronie 108

Poniższy kod zawiera przykłady podstawowych zadań klienta.

Informacje pokrewne

spawn()
bind()
socket()
listen()
close()
accept()
send()
recv()

Przykład: program serwera do tworzenia puli wielu procesów roboczych accept():

Przykład ten ilustruje, jak używać modeli wielu funkcji API accept() do tworzenia puli zadań procesów roboczych.

Uwaga: Korzystając z przykładowego kodu, użytkownik akceptuje warunki określone w temacie “Informacje dotyczące licencji na kod” na stronie 192.

```

/*****
/* 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;

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

/*****/
/* Umożliwia ponowne użycie deskryptora gniazda */
/*****/
rc = setsockopt(listen_sd,
                SOL_SOCKET, SO_REUSEADDR,
                (char *)&on, sizeof(on));

if (rc < 0)
{
    perror("Niepowodzenie setsockopt()");
    close(listen_sd);
    exit(-1);
}

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

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

/*****/
/* Inicjowanie parametrów przed wejściem w pętlę */
/* */
/* Deskryptor gniazda nasłuchiwania jest odwzoro-*/
/* wywany na deskryptor 0 programu potomnego. */
/*****/

```



```

memset(&inherit, 0, sizeof(inherit));
spawn_argv[0] = NULL;
spawn_envp[0] = NULL;
spawn_fdmmap[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_fdmmap, &inherit,
                spawn_argv, spawn_envp);
    if (pid < 0)
    {
        perror("Niepowodzenie spawn()");
        close(listen_sd);
        exit(-1);
    }
    printf(" Proces roboczy = %d\n", pid);
}

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

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

```

Odsyłacze pokrewne

“Przykład: ogólny program klienta” na stronie 108
Poniższy kod zawiera przykłady podstawowych zadań klienta.

Przykład: procesy robocze dla wielu funkcji accept():

Przykład ten ilustruje użycie wielu funkcji API accept() do odebrania zadań procesu roboczego i wywołania serwera accept().

Uwaga: Korzystając z przykładowego kodu, użytkownik akceptuje warunki określone w temacie “Informacje dotyczące licencji na kod” na stronie 192.

```

/*****
/* 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 nasłuchiwanie przekazywany */
    /* jest do tego zadania procesu roboczego jako */
    /* parametr wiersza komend */
    /*****

```

```

listen_sd = 0;

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

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

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

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

```

Przykład: ogólny program klienta

Poniższy kod zawiera przykłady podstawowych zadań klienta.

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

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

- Serwer iteracyjny. Przykład programu znajduje się w sekcji Przykład: pisanie programu serwera iteracyjnego.
- Potomny serwer i proces roboczy. Przykład programu znajduje się w sekcji Przykład: używanie funkcji API `spawn()` do tworzenia procesów potomnych.

- Serwer `sendmsg()` i proces roboczy `rcvmsg()`. Przykład programu znajduje się w sekcji Przykład: program serwera używany dla funkcji `sendmsg()` i `rcvmsg()`.
- Konstrukcja z wielokrotną funkcją `accept()`. Przykład programu znajduje się w sekcji Przykład: program serwera do tworzenia puli wielu zadań `accept()` procesów roboczych.
- Nieblokujące operacje `we/wy` i funkcja `select()` Przykładowy program znajduje się w sekcji Przykład: nieblokujące operacje `we/wy` i funkcja `select()`.
- Serwer akceptujący połączenia od klientów IPv4 i IPv6. Przykład programu znajduje się w sekcji Przykład: akceptowanie połączeń od klientów IPv6 i IPv4.

Przebieg zdarzeń w gnieździe: ogólny program klienta

Poniższy program używa wywołań następujących funkcji:

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

Uwaga: Korzystając z przykładowego kodu, użytkownik akceptuje warunki określone w temacie “Informacje dotyczące licencji na kod” na stronie 192.

```

/*****
/* Przykład ogólnego klienta dla serwerów zorientowanych na połączenie */
*****/
#include <stdio.h>
#include <stdlib.h>
#include <sys/socket.h>
#include <netinet/in.h>

#define SERVER_PORT 12345

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

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

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

    /*****
    /* Połączenie z serwerem */
    *****/

```

```

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

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

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

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

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

```

Odsyłacze pokrewne

“Przykład: klient IPv4 lub IPv6” na stronie 80

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

“Przykłady: projekty aplikacji zorientowanych na połączenie” na stronie 86

W systemie iSeries istnieje wiele możliwości realizacji zorientowanego na połączenie serwera gniazd. Do utworzenia własnych programów zorientowanych na połączenie można użyć poniższych programów przykładowych.

“Przykład: pisanie programu serwera iteracyjnego” na stronie 87

Poniżej znajduje się przykład pojedynczego zadania serwera, które obsługuje wszystkie połączenia przychodzące. Po zakończeniu działania funkcji API accept() serwer obsługuje całą transakcję.

“Przykład: przekazywanie deskryptorów pomiędzy procesami” na stronie 96

Przykłady funkcji sendmsg() i recvmsg() pokazują, jak zaprojektować program serwera, który korzysta z tych funkcji do obsługi połączeń przychodzących.

“Przykład: program serwera używany dla funkcji `sendmsg()` i `recvmsg()`” na stronie 98

Przykład ten ilustruje, jak używać funkcji API `sendmsg()` do tworzenia puli zadań procesów roboczych.

“Przykład: używanie wielu funkcji API `accept()` do obsługi żądań przychodzących” na stronie 103

Przykłady te ilustrują sposób projektowania programu serwera, który używa modelu wielu funkcji `accept()` do obsługi żądań połączeń przychodzących.

“Przykład: program serwera do tworzenia puli wielu procesów roboczych `accept()`” na stronie 105

Przykład ten ilustruje, jak używać modeli wielu funkcji API `accept()` do tworzenia puli zadań procesów roboczych.

“Przykład: korzystanie z asynchronicznych operacji we/wy”

Aplikacja tworzy port we/wy zakończenia za pomocą funkcji API `QsoCreateIOCompletionPort()`.

“Przykład: nieblokujące operacje we/wy i funkcja `select()`” na stronie 151

Poniższy program przykładowy wykorzystuje nieblokujące operacje we/wy i funkcję API `select()`.

Informacje pokrewne

`socket()`

`connect()`

`close()`

`send()`

`recv()`

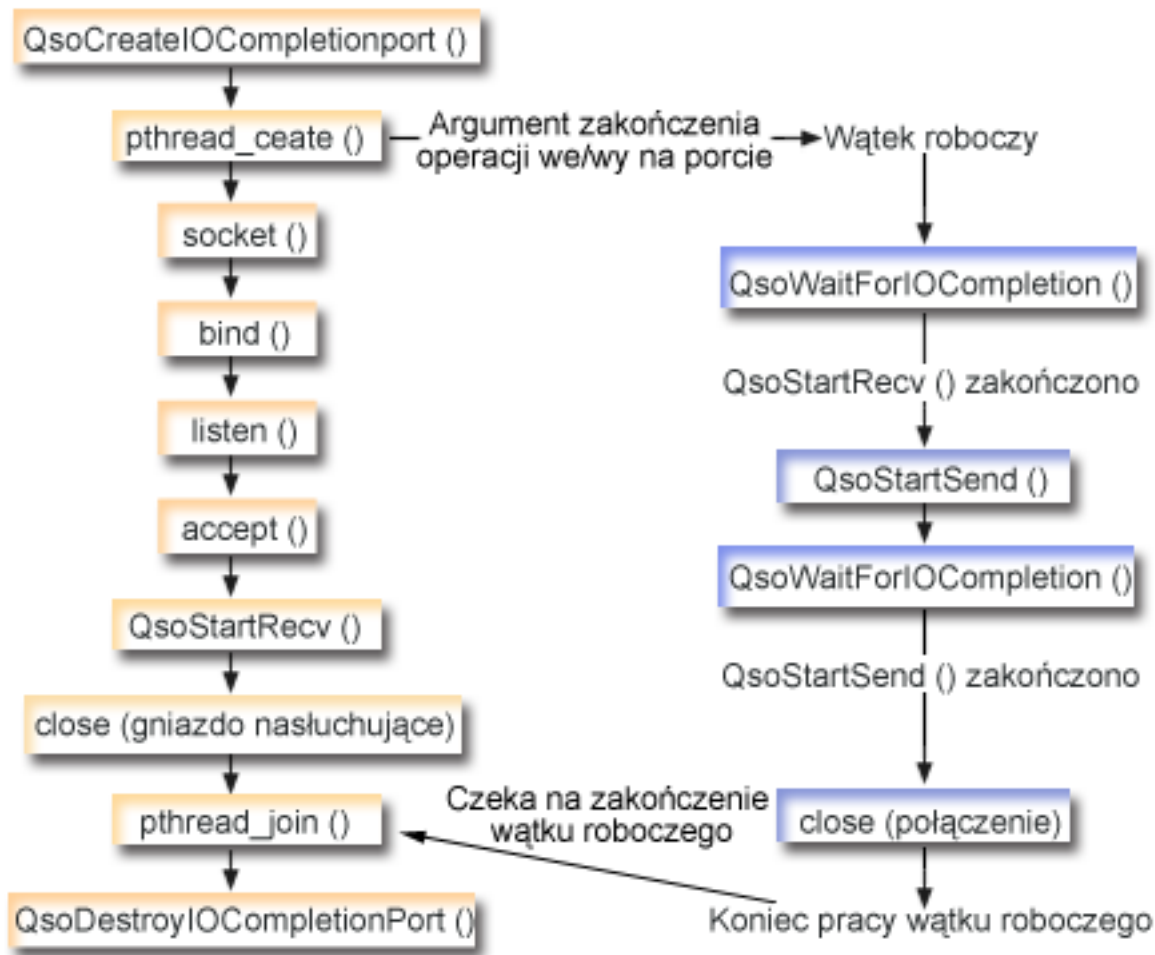
Przykład: korzystanie z asynchronicznych operacji we/wy

Aplikacja tworzy port we/wy zakończenia za pomocą funkcji API `QsoCreateIOCompletionPort()`.

Funkcja ta zwraca uchwyt, którego można użyć do zaplanowania i oczekiwania na zakończenie żądań asynchronicznych operacji we/wy. Następnie aplikacja uruchamia funkcję operacji wejścia lub operacji wyjścia, podając uchwyt portu zakończenia operacji we/wy. Kiedy operacje we/wy zostają zakończone, informacje o statusie i zdefiniowany przez aplikację uchwyt zostają zapisane w podanym porcie we/wy zakończenia. Zapisanie do portu we/wy zakończenia uaktywnia dokładnie jeden z 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,
- zdefiniowany przez aplikację uchwyt, który został 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 zakończyła się, a uchwyt aplikacji został przekazany, wątek procesu roboczego określa następny krok w celu zakończenia połączenia klienta. Wątki procesów roboczych, które przetwarzają te zakończone operacje asynchroniczne, są w stanie obsłużyć wiele różnych żądań klienta i nie są powiązane tylko z jednym z nich. Ponieważ kopiowanie z i do buforów użytkowników odbywa się asynchronicznie względem procesów serwera, czasy oczekiwania dla żądań klientów skracają się. Cecha ta może być szczególnie korzystna w systemach wieloprocessorowych.



Przebieg zdarzeń w gnieździe: asynchroniczny serwer we/wy

Na poniższej ilustracji wyjaśniono sekwencję wywołań funkcji gniazd. Opisano także relacje pomiędzy aplikacją serwera a przykładowym procesem roboczym. Każdy zbiór wywołań zawiera odsyłacze do uwag dotyczących użycia poszczególnych funkcji API. Aby uzyskać szczegółowe informacje dotyczące użycia tych funkcji API, należy użyć poniższych odsyłaczy. Diagram ten przedstawia wywołania funkcji gniazd w poniższej przykładowej aplikacji. Serwer ten może współpracować z przykładowym klientem ogólnym.

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

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

5. W pewnym momencie żądanie klienta dociera asynchronicznie do procesu serwera. System operacyjny gniazd wczytuje dostarczony bufor użytkownika i wysyła zakończone żądanie `QsoStartRecv()` do określonego portu we/wy zakończenia. Jeden wątek procesu roboczego zostaje uaktywniony i kontynuuje przetwarzanie tego żądania.
6. Wątek procesu roboczego wyodrębnia deskryptor gniazda z uchwytu zdefiniowanego przez aplikację i odsyła odebrane dane z powrotem do klienta, wykonując operację `QsoStartSend()`.

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

Uwaga: Korzystając z przykładowego kodu, użytkownik akceptuje warunki określone w temacie “Informacje dotyczące licencji na kod” na stronie 192.

```
#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 SERVPOR 12345

void *workerThread(void *arg);

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

int main() {
int listen_sd, client_sd, rc;
int on = 1, ioCompPort;
pthread_t thr;
void *status;
char buffer[BufferLength];
struct sockaddr_in serveraddr;
Qso_OverlappedIO_t ioStruct;

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

/*****
/* Tworzy wątek procesu roboczego w celu
/* przetwarzania wszystkich żądań klienta
*****/
```

```

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

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

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

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

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

/*****
/* Ustawia kolejkę (backlog) nasłuchiwania. */
/*****
if ((rc = listen(listen_sd, 10)) < 0)
{
    perror("Niepowodzenie listen()");
    QsoDestroyIOCompletionPort(ioCompPort);
    close(listen_sd);
    exit(-1);
}

```



```

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

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

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

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

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

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

rc = QsoStartRecv(client_sd, ioCompPort, &ioStruct);
if (rc == -1)
{
    perror("Niepowodzenie QsoStartRecv()");
    QsoDestroyIOCompletionPort(ioCompPort);
    close(listen_sd);
    close(client_sd);
    exit(-1);
}

/*****
/* Zamknięcie gniazda nasłuchującego serwera */
*****/
close(listen_sd);

```

```

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

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

/*****
/*
/* Nazwa funkcji: workerThread
/*
/*
/* Nazwa opisowa: Przetwarzanie połączenia klienta.
/*
/*****
void *workerThread(void *arg)
{
    struct timeval waitTime;
    int ioCompPort, clientfd;
    Qso_OverlappedIO_t ioStruct;
    int rc, tID;
    pthread_t thr;
    pthread_id_np_t t_id;
    t_id = pthread_getthreadid_np();
    tID = t_id.intId.lo;

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

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

    /*****
    /* Uzyskuje deskryptor gniazda powiązanego z */
    /* połączeniem klienta.
    /*

```

```

/*****/
clientfd = *((int *) &ioStruct.descriptorHandle);

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

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

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

```

Pojęcia pokrewne

“Asynchroniczne operacje we/wy” na stronie 43

Funkcje API asynchronicznych operacji we/wy udostępniają metodę dla wątków modelu klient serwer, umożliwiającą realizację współbieżnych operacji we/wy z efektywnym wykorzystaniem pamięci.

Odsyłacze pokrewne

“Zalecenia dotyczące projektowania aplikacji używających gniazd” na stronie 83

Aby projektować aplikacje używające gniazd, należy określić wymagania funkcjonalne, cele i potrzeby tej aplikacji.

“Przykłady: projekty aplikacji zorientowanych na połączenie” na stronie 86

W systemie iSeries istnieje wiele możliwości realizacji zorientowanego na połączenie serwera gniazd. Do utworzenia własnych programów zorientowanych na połączenie można użyć poniższych programów przykładowych.

“Przykład: ogólny program klienta” na stronie 108

Poniższy kod zawiera przykłady podstawowych zadań klienta.

“Przykład: używanie sygnałów z blokującymi funkcjami API gniazd” na stronie 163

Sygnały powiadamiają o zablokowaniu procesu lub aplikacji.

Informacje pokrewne

QsoCreateIOCompletionPort()

pthread_create

QsoWaitForIoCompletionPort()

QsoStartRecv()

QsoStartAccept()

QsoStartSend()

QsoDestroyIOCompletionPort()

Przykład: nawiązywanie chronionych połączeń

Chroniony serwer i klienta można utworzyć przy użyciu funkcji API Global Secure ToolKit (GSKit) lub SSL_.

Funkcje GSKit API są preferowane ponieważ obsługują je wszystkie systemy IBM, podczas gdy funkcje SSL_API występują tylko w systemie i5/OS. Oba zestawy funkcji API gniazd chronionych mają kody powrotu, które pomagają zidentyfikować błędy powstałe podczas nawiązywania połączeń chronionych.

Pojęcia pokrewne

“Komunikaty z kodami błędów funkcji API gniazd chronionych” na stronie 51

Aby uzyskać informacje o komunikatach kodów błędów gniazda chronionego, należy postępować zgodnie ze wskazówkami.

Przykład: chroniony serwer GSKit z asynchronicznym odbieraniem danych

Poniższego programu przykładowego można użyć do tworzenia chronionego serwera przy użyciu funkcji API Global Secure ToolKit (GSKit).

Serwer otwiera gniazdo, przygotowuje chronione środowisko, akceptuje i przetwarza żądania połączenia, wymienia dane z klientem i kończy sesję. Klient również otwiera gniazdo, ustanawia chronione środowisko, nawiązuje połączenie z serwerem i żąda połączenia chronionego, wymienia dane z serwerem i kończy sesję. Poniższy diagram i towarzyszący mu opis ilustrują przebieg zdarzeń pomiędzy serwerem a klientem.

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

Przebieg zdarzeń w gnieździe: chroniony serwer używający asynchronicznych funkcji API_

Na poniższej ilustracji wyjaśniono sekwencję wywołań funkcji gniazd. Opisano także relacje pomiędzy aplikacją serwera a aplikacją klienta.

1. Funkcja QsoCreateIOCompletionPort() tworzy port we/wy zakończenia.

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

Uwaga: Aby uzgadnianie SSL się powiodło, program serwera musi okazać certyfikat. Ponadto serwer musi mieć dostęp do klucza prywatnego powiązanego z certyfikatem serwera i zbioru bazy danych kluczy, w którym przechowywany jest certyfikat. W niektórych przypadkach klient także musi okazać certyfikat podczas przetwarzania uzgadniania SSL. Dotyczy to sytuacji, w której serwer, z którym łączy się klient, ma włączone uwierzytelnianie klienta. Wywołania funkcji API `gsk_attribute_set_buffer(GSK_OS400_APPLICATION_ID)` lub `gsk_attribute_set_buffer(GSK_KEYRING_FILE)` identyfikują (różnymi metodami) zbiór bazy danych kluczy, z którego uzyskano klucz prywatny, i certyfikat użyte podczas uzgadniania.

10. Funkcja `gsk_secure_soc_startRecv()` Inicjuje asynchroniczną operację odbioru podczas bezpiecznej sesji.
11. Funkcja `pthread_join` synchronizuje programy serwera i procesu roboczego. Oczekuje ona na zakończenie wątku, odłącza go, a następnie zwraca status wyjścia wątku do serwera.
12. Funkcja `gsk_secure_soc_close()` kończy sesję chronioną.
13. Funkcja `gsk_environment_close()` zamyka środowisko SSL.
14. Funkcja `close()` kończy gniazdo nasłuchujące.
15. Funkcja `close()` kończy gniazdo, które przyjęło połączenie od klienta.
16. Funkcja `QsoDestroyIOCompletionPort()` niszczy port zakończenia.

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

1. Utworzony przez aplikację serwera wątek procesu roboczego oczekuje na wysłane przez serwer przychodzące żądanie klienta, aby obsłużyć dane klienta za pomocą wywołania funkcji `gsk_secure_soc_startRecv()`. Funkcja `QsoWaitForIOCompletionPort()` oczekuje na przekazany jej przez serwer porcie zakończenia we/wy.
2. Po otrzymaniu żądania klienta funkcja `gsk_attribute_get_numeric_value()` pobiera deskryptor gniazda powiązany z sesją chronioną.
3. Funkcja `gsk_secure_soc_write()` wysyła do klienta komunikat w ramach sesji chronionej.

Uwaga: Korzystając z przykładowego kodu, użytkownik akceptuje warunki określone w temacie “Informacje dotyczące licencji na kod” na stronie 192.

```

/* Program asynchronicznego serwera GSK używający identyfikatora aplikacji */

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

/* Użyj następującej komendy, aby utworzyć program  */
/* skonsolidowany:                                  */
/* CRTBNDC PGM(PROG/GSKSERVa)                       */
/*          SRCFILE(PROG/CSRC)                       */
/*          SRCMBR(GSKSERVa)                         */

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

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

    struct sockaddr_in address;
    int buf_len, on = 1, rc = 0;
    int sd = -1, lsd = -1, al = -1, ioCompPort = -1;
    int successFlag = FALSE;
    char buff[1024];

```

```

pthread_t thr;
void *status;
Qso_OverlappedIO_t ioStruct;

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

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

    /* ustawia ID aplikacji */
    rc = errno = 0;
    rc = gsk_attribute_set_buffer(my_env_handle,
                                GSK_OS400_APPLICATION_ID,
                                "MY_SERVER_APP",
                                13);

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

    /* ustawia tę stronę jako serwer */
    rc = errno = 0;
    rc = gsk_attribute_set_enum(my_env_handle,
                                GSK_SESSION_TYPE,
                                GSK_SERVER_SESSION);

    if (rc != GSK_OK)
    {
        printf("Niepowodzenie gsk_attribute_set_enum() z kodem powrotu = %d kod błędu = %d.\n",

```

```

        rc,errno);
    printf("kod powrotu %d oznacza %s\n", rc, gsk_strerror(rc));
    break;
}

/* Protokoły SSL_V2, SSL_V3 i TLS_V1 są włączone domyślnie. */
/* W tym przykładzie wyłączymy protokół SSL_V2. */
rc = errno = 0;
rc = gsk_attribute_set_enum(my_env_handle,
                           GSK_PROTOCOL_SSLV2,
                           GSK_PROTOCOL_SSLV2_OFF);

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

/* Określa, jakiego pakietu szyfrującego użyć. Domyślnie włączona jest */
/* domyślna lista szyfrowania. W tym przykładzie użyjemy jednego. */
rc = errno = 0;
rc = gsk_attribute_set_buffer(my_env_handle,
                              GSK_V3_CIPHER_SPECS,
                              "05", /* SSL_RSA_WITH_RC4_128_SHA */
                              2);

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

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

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

/* ustawienie gniazda do natychmiastowego ponownego użycia */
rc = setsockopt(lfd, SOL_SOCKET,
                SO_REUSEADDR,
                (char *)&on,
                sizeof(on));

if (rc < 0)
{
    perror("Niepowodzenie setsockopt()");
    break;
}

/* powiązanie do adresu lokalnego serwera */
memset((char *) &address, 0, sizeof(address));
address.sin_family = AF_INET;

```



```

address.sin_port = 13333;
address.sin_addr.s_addr = 0;
rc = bind(lsd, (struct sockaddr *) &address, sizeof(address));
if (rc < 0)
{
    perror("Niepowodzenie bind()");
    break;
}

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

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

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

/* powiązanie gniazda z sesją chronioną */
rc=errno=0;
rc = gsk_attribute_set_numeric_value(my_session_handle,
                                     GSK_FD,
                                     sd);

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

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

/*****
/* Wywołanie gsk_secure_soc_startRecv() w      */
/* odebrania żądania od klienta.              */
/* Uwaga:                                       */
/* postFlag == podczas odczytywania żądania   */
/* powinien być zapisany do portu we/wy      */

```

```

/* zakończenia, nawet jeśli żądanie jest */
/* dostępne natychmiast. Wątek roboczy */
/* przetworzy żądanie klienta. */
/*****/
/*****/
/* Inicjuje strukturę Qso_OverlappedIO_t - */
/* w polach zastrzeżonych muszą być */
/* szesnastkowe 00. */
/*****/
memset(&ioStruct, '\0', sizeof(ioStruct));
memset((char *) buff, 0, sizeof(buff));
ioStruct.buffer = buff;
ioStruct.bufferLength = sizeof(buff);

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

rc = gsk_secure_soc_startRecv(my_session_handle,
                             ioCompPort,
                             &ioStruct);
if (rc != GSK_AS400_ASYNCHRONOUS_RECV)
{
    printf("Kod powrotu gsk_secure_soc_startRecv() = %d, numer błędu = %d.\n", rc,errno);
    printf("kod powrotu %d oznacza %s\n", rc, gsk_strerror(rc));
    break;
}

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

/*****/
/* 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 pthread_join()");
}
} while(FALSE);

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

/* wyłączenie środowiska SSL */

```

```

if (my_env_handle != NULL)
    gsk_environment_close(&my_env_handle);

/* zamknięcie gniazda nasłuchującego */
if (lfd > -1)
    close(lfd);
/* 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 uchwyt */
/* clientfd i session. */
*****/
void *workerThread(void *arg)
{
    struct timeval waitTime;
    int ioCompPort = -1, clientfd = -1;
    Qso_OverlappedIO_t ioStruct;
    int rc, tID;
    int amtWritten;
    gsk_handle client_session_handle = NULL;
    pthread_t thr;
    pthread_id_np_t t_id;
    t_id = pthread_getthreadid_np();
    tID = t_id.intId.lo;
    /*****/
    /* Port we/wy zakończenia jest przekazywany */
    /* do tej procedury.*/
    /*****/
    ioCompPort = *(int *)arg;
    /*****/
    /* Czekaj przy dostarczonej porcie we/wy */
    /* zakończenia na żądanie klienta. */
    /*****/
    waitTime.tv_sec = 500;
    waitTime.tv_usec = 0;
    rc = QsoWaitForIOCompletion(ioCompPort, &ioStruct, &waitTime);
    if ((rc == 1) &&
        (ioStruct.returnValue == GSK_OK) &&
        (ioStruct.operationCompleted == GSKSECURESOCSTARTRECV))
    /*****/
    /* Odebrano żądanie klienta. */
    /*****/
    ;
    else
    {
        perror("Niepowodzenie QsoWaitForIOCompletion()/gsk_secure_soc_startRecv()");
        printf("ioStruct.returnValue = %d.\n", ioStruct.returnValue);
        return __VOID(Failure);
    }
}

```

```

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

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

/* pobranie gniazda powiązanego z sesją chronioną */
rc=errno=0;
rc = gsk_attribute_get_numeric_value(client_session_handle,
                                     GSK_FD,
                                     &clientfd);

if (rc != GSK_OK)
{
  printf("Kod powrotu gsk_attribute_get_numeric_value() = %d, numer błędu = %d.\n",
        rc,errno);
  printf("kod powrotu %d oznacza %s\n", rc, gsk_strerror(rc));
  return __VOID(Failure);
}

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

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

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

```

Pojęcia pokrewne

“Funkcje API Global Secure ToolKit (GSKit)” na stronie 47

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

Odsyłacze pokrewne

“Przykład: chroniony klient używający funkcji API Global Secure ToolKit (GSKit)” na stronie 136

Poniższy kod stanowi przykład klienta używającego funkcji API GSKit.

“Przykład: chroniony serwer GSKit z asynchronicznym uzgadnianiem” na stronie 127

Funkcja ta umożliwia tworzenie chronionych aplikacji serwera, które mogą obsługiwać żądania w sposób asynchroniczny.

Informacje pokrewne

```
QsoCreateIOCompletionPort()
pthread_create
QsoWaitForIoCompletionPort()
QsoDestroyIOCompletionPort()
bind()
socket()
listen()
close()
accept()
gsk_environment_open()
gsk_attribute_set_buffer
gsk_attribute_set_enum()
gsk_environment_init()
gsk_secure_soc_open()
gsk_attribute_set_numeric_value()
gsk_secure_soc_init()
gsk_secure_soc_startRecv()
pthread_join
gsk_secure_soc_close()
gsk_environment_close()
gsk_attribute_get_numeric_value ()
gsk_secure_soc_write ()
```

Przykład: chroniony serwer GSKit z asynchronicznym uzgadnianiem

Funkcja ta umożliwia tworzenie chronionych aplikacji serwera, które mogą obsługiwać żądania w sposób asynchroniczny.

Poniższy kod stanowi przykład, jak można użyć tej funkcji. Jest on podobny do przykładu chronionego serwera GSKit z asynchronicznym odbieraniem danych, ale do rozpoczęcia chronionej sesji używa opisanej tu funkcji API.

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

Część diagramu dotycząca klienta GSKit znajduje się w sekcji `xgskclient.dita#xgskclient/gskclgraphic`.

Przebieg zdarzeń w gnieździe: chroniony serwer GSKit z asynchronicznym uzgadnianiem

Diagram ten przedstawia wywołania funkcji gniazd w poniższej przykładowej aplikacji.

1. Funkcja `QsoCreateIOCompletionPort()` tworzy port we/wy zakończenia.
2. Funkcja `pthread_create` tworzy wątek procesu roboczego, który będzie przetwarzał wszystkie żądania klienta. Wątek procesu roboczego będzie oczekiwał na nadejście żądań od klienta na utworzonym właśnie porcie we/wy zakończenia.
3. Wywołanie funkcji `gsk_environment_open()` w celu uzyskania uchwytu środowiska SSL.
4. Przynajmniej jedno wywołanie funkcji `gsk_attribute_set_xxxxx()` w celu ustawienia atrybutów środowiska SSL. Minimum to wywołanie funkcji `gsk_attribute_set_buffer()` w celu ustawienia wartości `GSK_OS400_APPLICATION_ID` lub wartości `GSK_KEYRING_FILE`. Należy ustawić tylko jedną z tych wartości. Zalecane jest użycie wartości `GSK_OS400_APPLICATION_ID`. Ponadto należy ustawić typ aplikacji (klient lub serwer), `GSK_SESSION_TYPE`, używając funkcji `gsk_attribute_set_enum()`.

5. Wywołanie funkcji `gsk_environment_init()` w celu zainicjowania tego środowiska do przetwarzania SSL i określenia informacji o ochronie dla wszystkich sesji SSL, które będą uruchamiane w tym środowisku.
6. Funkcja `socket` tworzy deskryptor gniazda. Następnie serwer wywołuje standardowy zestaw funkcji gniazd: `bind()`, `listen()` i `accept()`, w celu umożliwienia akceptowania przychodzących żądań połączenia.
7. Funkcja `gsk_secure_soc_open()` uzyskuje pamięć dla bezpiecznej sesji, ustawia domyślne wartości atrybutów i zwraca uchwyt, który musi zostać zapisany i użyty podczas następnych wywołań funkcji związanych z chronioną sesją.
8. Przynajmniej jedno wywołanie funkcji `gsk_attribute_set_xxxxx()` w celu ustawienia atrybutów bezpiecznej sesji. Minimum to wywołanie funkcji `gsk_attribute_set_numeric_value()` w celu powiązania określonego gniazda z tą bezpieczną sesją.
9. Funkcja `gsk_secure_soc_startInit()` uruchamia asynchroniczną negocjację bezpiecznej sesji, używając zestawu atrybutów dla środowiska SSL i bezpiecznej sesji. W tym miejscu sterowanie jest z powrotem przekazywane do programu. Po zakończeniu procesu uzgadniania port zakończenia zostaje zaktualizowany razem z rezultatami. Wątek może kontynuować przetwarzanie; jednakże dla uproszczenia program oczekuje na zakończenie pracy wątku procesu roboczego.

Uwaga: Aby uzgadnianie SSL się powiodło, program serwera musi okazać certyfikat. Ponadto serwer musi mieć dostęp do klucza prywatnego powiązanego z certyfikatem serwera i zbioru bazy danych kluczy, w którym przechowywany jest certyfikat. W niektórych przypadkach klient także musi okazać certyfikat podczas przetwarzania uzgadniania SSL. Dotyczy to sytuacji, w której serwer, z którym łączy się klient, ma włączone uwierzytelnianie klienta. Wywołania funkcji API `gsk_attribute_set_buffer(GSK_OS400_APPLICATION_ID)` lub `gsk_attribute_set_buffer(GSK_KEYRING_FILE)` identyfikują (różnymi metodami) zbiór bazy danych kluczy, z którego uzyskano klucz prywatny, i certyfikat użyte podczas uzgadniania.

10. Funkcja `pthread_join` synchronizuje programy serwera i procesu roboczego. Oczekuje ona na zakończenie wątku, odłącza go, a następnie zwraca status wyjścia wątku do serwera.
11. Funkcja `gsk_secure_soc_close()` kończy sesję chronioną.
12. Funkcja `gsk_environment_close()` zamyka środowisko SSL.
13. Funkcja `close()` kończy gniazdo nasłuchujące.
14. Funkcja `close()` kończy gniazdo, które przyjęło połączenie od klienta.
15. Funkcja `QsoDestroyIOCompletionPort()` niszczy port zakończenia.

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

1. Utworzony przez aplikację serwera wątek procesu roboczego oczekuje na wysłane przez serwer przychodzące żądanie klienta, aby obsłużyć dane klienta. Funkcja `QsoWaitForIOCompletionPort()` oczekuje na przekazany jej przez serwer porcie zakończenia `we/wy`. Wywołanie to oczekuje na zakończenie działania funkcji `gsk_secure_soc_startInit()`.
2. Po otrzymaniu żądania klienta funkcja `gsk_attribute_get_numeric_value()` pobiera deskryptor gniazda powiązany z sesją chronioną.
3. Funkcja `gsk_secure_soc_read()` odbiera od klienta komunikat w ramach sesji chronionej.
4. Funkcja `gsk_secure_soc_write()` wysyła do klienta komunikat w ramach sesji chronionej.

Uwaga: Korzystając z przykładowego kodu, użytkownik akceptuje warunki określone w temacie “Informacje dotyczące licencji na kod” na stronie 192.

```

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

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

```

```

/* Użyj następującej komendy, aby utworzyć program */
/* skonsolidowany: */
/* CRTBNDC PGM(MYLIB/GSKSERVSI) */
/* SRCFILE(MYLIB/CSRC) */
/* SRCMBR(GSKSERVSI) */

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

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

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

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

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

```

```

rc = pthread_create(&thr, NULL, workerThread, &ioCompPort);
if (rc < 0)
{
    perror("Niepowodzenie pthread_create()");
    break;
}

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

/* ustawia ID aplikacji */
rc = errno = 0;
rc = gsk_attribute_set_buffer(my_env_handle,
                              GSK_OS400_APPLICATION_ID,
                              "MY_SERVER_APP",
                              13);

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

/* ustawia tę stronę jako serwer */
rc = errno = 0;
rc = gsk_attribute_set_enum(my_env_handle,
                            GSK_SESSION_TYPE,
                            GSK_SERVER_SESSION);

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

/* Protokoły SSL_V2, SSL_V3 i TLS_V1 są włączone domyślnie. */
/* W tym przykładzie wyłączymy protokół SSL_V2. */
rc = errno = 0;
rc = gsk_attribute_set_enum(my_env_handle,
                            GSK_PROTOCOL_SSLV2,
                            GSK_PROTOCOL_SSLV2_OFF);

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

/* Określa, jakiego pakietu szyfrującego użyć. Domyślnie włączona jest */
/* domyślna lista szyfrowania. W tym przykładzie użyjemy jednego. */
rc = errno = 0;
rc = gsk_attribute_set_buffer(my_env_handle,
                              GSK_V3_CIPHER_SPECS,
                              "05", /* SSL_RSA_WITH_RC4_128_SHA */
                              2);

```



```

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

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

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

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

/* powiązanie do adresu lokalnego serwera */
memset((char *) &address, 0, sizeof(address));
address.sin_family = AF_INET;
address.sin_port = 13333;
address.sin_addr.s_addr = 0;
printf("bind()\n");
rc = bind(lfd, (struct sockaddr *) &address, sizeof(address));
if (rc < 0)
{
    perror("Niepowodzenie bind()");
    break;
}

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

/* akceptacja przychodzącego połączenia klienta */
al = sizeof(address);
printf("accept()\n");
sd = accept(lfd, (struct sockaddr *) &address, &al);
if (sd < 0)

```

```

{
perror("Niepowodzenie accept()");
break;
}

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

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

/*****/
/* Wywołanie gsk_secure_soc_startInit() w */
/* celu asynchronicznego przetworzenia */
/* uzgadniania SSL */
/*****/
/*****/
/* Inicjuje strukturę Qso_OverlappedIO_t - */
/* w polach zastrzeżonych muszą być */
/* szesnastkowe 00. */
/*****/
memset(&ioStruct, '\0', sizeof(ioStruct));

/*****/
/* Przechowuje uchwyt sesji w polu */
/* descriptorHandle Qso_OverlappedIO_t. */
/* Obszar ten jest używany do przechowywania */
/* informacji określających stan połączenia */
/* klienta. Pole descriptorHandle jest */
/* definiowane jako (void *), aby serwer mógł */
/* w razie potrzeby obsłużyć jak najszerszy */
/* zakres stanów połączenia klienta. */
/*****/
ioStruct.descriptorHandle = my_session_handle;

/* inicjowanie uzgodnienia SSL */
rc = errno = 0;
printf("gsk_secure_soc_startInit()\n");
rc = gsk_secure_soc_startInit(my_session_handle, ioCompPort, &ioStruct);
if (rc != GSK_OS400_ASYNCHRONOUS_SOC_INIT)
{
printf("Kod powrotu gsk_secure_soc_startInit() = %d, numer błędu = %d.\n",rc,errno);
printf("kod powrotu %d oznacza %s\n", rc, gsk_strerror(rc));
break;
}
else
printf("Funkcja gsk_secure_soc_startInit odebrała GSK_OS400_ASYNCHRONOUS_SOC_INIT\n");

/*****/

```

```

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

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

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

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

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

/* zamknięcie gniazda nasłuchującego */
if (lfd > -1)
    close(lfd);
/* 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 uchwyt */
/* clientfd i session. */
/*****/
void *workerThread(void *arg)
{
    struct timeval waitTime;
    int ioCompPort, clientfd;
    Qso_OverlappedIO_t ioStruct;
    int rc, tID;
    int amtWritten, amtRead;
    char buff[1024];

```

```

gsk_handle client_session_handle;
pthread_t thr;
pthread_id_np_t t_id;
t_id = pthread_getthreadid_np();
tID = t_id.intId.lo;
/*****/
/* Port we/wy zakończenia jest przekazywany */
/* do tej procedury. */
/*****/
ioCompPort = *(int *)arg;
/*****/
/* Czeka przy dostarczonym porcie we/wy */
/* na zakończenie uzgadniania SSL. */
/*****/
waitTime.tv_sec = 500;
waitTime.tv_usec = 0;

sleep(4);
printf("QsoWaitForIOCompletion()\n");
rc = QsoWaitForIOCompletion(ioCompPort, &ioStruct, &waitTime);
if ((rc == 1) &&
    (ioStruct.returnValue == GSK_OK) &&
    (ioStruct.operationCompleted == GSKSECURESOCSTARTINIT))
/*****/
/* Uzgadnianie SSL zostało zakończone. */
/*****/
;
else
{
    printf("Niepowodzenie QsoWaitForIOCompletion()/gsk_secure_soc_startInit().\n");
    printf("rc == %d, returnValue = %d, operationCompleted = %d\n",
        rc, ioStruct.returnValue, ioStruct.operationCompleted);
    perror("Niepowodzenie QsoWaitForIOCompletion()/gsk_secure_soc_startInit()");
    return __VOID(Failure);
}

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

/* pobranie gniazda powiązanego z sesją chronioną */
rc=errno=0;
printf("gsk_attribute_get_numeric_value()\n");
rc = gsk_attribute_get_numeric_value(client_session_handle,
    GSK_FD,
    &clientfd);

if (rc != GSK_OK)
{
    printf("Kod powrotu gsk_attribute_get_numeric_value() = %d, numer błędu = %d.\n",
        rc,errno);
    printf("kod powrotu %d oznacza %s\n", rc, gsk_strerror(rc));
    return __VOID(Failure);
}
/* Funkcja memset zeruje bufor szesnastkowo */
memset((char *) buff, 0, sizeof(buff));
amtRead = 0;
/* odebranie komunikatu od klienta w ramach bezpiecznej sesji */
printf("gsk_secure_soc_read()\n");
rc = gsk_secure_soc_read(client_session_handle,
    buff,
    sizeof(buff),
    &amtRead);

if (rc != GSK_OK)
{

```

```

    printf("Kod powrotu gsk_secure_soc_read() = %d, numer błędu = %d.\n",rc,errno);
    printf("kod powrotu %d oznacza %s\n", rc, gsk_strerror(rc));
    return;
}

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

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

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

```

Pojęcia pokrewne

“Funkcje API Global Secure Toolkit (GSKit)” na stronie 47

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

Odsyłacze pokrewne

“Przykład: chroniony serwer GSKit z asynchronicznym odbieraniem danych” na stronie 118

Poniższego programu przykładowego można użyć do tworzenia chronionego serwera przy użyciu funkcji API Global Secure Toolkit (GSKit).

“Przykład: chroniony klient używający funkcji API Global Secure Toolkit (GSKit)” na stronie 136

Poniższy kod stanowi przykład klienta używającego funkcji API GSKit.

Informacje pokrewne

QsoCreateIOCompletionPort()

pthread_create

QsoWaitForIoCompletionPort()

QsoDestroyIOCompletionPort()

bind()

socket()

listen()

close()

accept()

```
gsk_environment_open()
gsk_attribute_set_buffer
gsk_attribute_set_enum()
gsk_environment_init()
gsk_secure_soc_open()
gsk_attribute_set_numeric_value()
gsk_secure_soc_init()
pthread_join
gsk_secure_soc_close()
gsk_environment_close()
gsk_attribute_get_numeric_value ()
gsk_secure_soc_write ()
gsk_secure_soc_startInit()
gsk_secure_soc_read ()
```

Przykład: chroniony klient używający funkcji API Global Secure Toolkit (GSKit)

Poniższy kod stanowi przykład klienta używającego funkcji API GSKit.

Poniższa ilustracja przedstawia wywołania funkcji w kliencie chronionym używającym funkcji API GSKit:

Przebieg zdarzeń w gnieździe: klient GSKit

Diagram ten przedstawia wywołania funkcji gniazd w poniższej przykładowej aplikacji. Klienta tego można używać z przykładowym serwerem GSKit i przykładowym chronionym serwerem GSKit z uzgadnianiem asynchronicznym.

1. Funkcja `gsk_environment_open()` jest wywoływana w celu uzyskania uchwytu środowiska SSL.
2. Przynajmniej jedno wywołanie funkcji `gsk_attribute_set_xxxxx()` w celu ustawienia atrybutów środowiska SSL. Minimum to wywołanie funkcji `gsk_attribute_set_buffer()` w celu ustawienia wartości `GSK_OS400_APPLICATION_ID` lub wartości `GSK_KEYRING_FILE`. Należy ustawić tylko jedną z tych wartości. Zalecane jest użycie wartości `GSK_OS400_APPLICATION_ID`. Ponadto należy ustawić typ aplikacji (klient lub serwer), `GSK_SESSION_TYPE`, używając funkcji `gsk_attribute_set_enum()`.
3. Wywołanie funkcji `gsk_environment_init()` w celu zainicjowania tego środowiska do przetwarzania SSL i określenia informacji o ochronie dla wszystkich sesji SSL, które będą uruchamiane w tym środowisku.
4. Funkcja `socket()` zwraca deskryptor gniazda odpowiadający punktowi końcowemu. Następnie klient wywołuje funkcję `connect()` w celu połączenia się z aplikacją serwera.
5. Funkcja `gsk_secure_soc_open()` uzyskuje pamięć dla bezpiecznej sesji, ustawia domyślne wartości atrybutów i zwraca uchwyt, który musi zostać zapisany i użyty podczas następnych wywołań funkcji związanych z chronioną sesją.
6. Funkcja `gsk_attribute_set_numeric_value()` przypisuje konkretne gniazdo do tej sesji chronionej.
7. Funkcja `gsk_secure_soc_init()` uruchamia asynchroniczną negocjację bezpiecznej sesji, używając zestawu atrybutów dla środowiska SSL i bezpiecznej sesji.
8. Funkcja `gsk_secure_soc_write()` zapisuje dane w sesji chronionej do wątku procesu roboczego.

Uwaga: W przykładzie serwera GSKit funkcja ta zapisuje dane do wątku procesu roboczego po zakończeniu działania funkcji `gsk_secure_soc_startRecv()`. W przykładzie serwera asynchronicznego zapisuje ona dane do zakończonej funkcji `gsk_secure_soc_startInit()`.

9. Funkcja `gsk_secure_soc_read()` odbiera komunikat od wątku procesu roboczego przy użyciu sesji chronionej.
10. Funkcja `gsk_secure_soc_close()` kończy sesję chronioną.
11. Funkcja `gsk_environment_close()` zamyka środowisko SSL.

12. Funkcja close() kończy połączenie.

Uwaga: Korzystając z przykładowego kodu, użytkownik akceptuje warunki określone w temacie “Informacje dotyczące licencji na kod” na stronie 192.

```
/* Program serwera GSK używający ID aplikacji */

/* Program zakłada, że identyfikator aplikacji został */
/* zarejestrowany, a certyfikat został przypisany */
/* do identyfikatora aplikacji. */
/* */
/* Brak parametrów, trochę komentarzy i wiele wartości */
/* wpisanych w kodzie, aby przykład był prosty. */

/* Użyj następującej komendy, aby utworzyć program */
/* skonsolidowany: */
/* CRTBND CPG(MYLIB/GSKCLIENT) */
/* SRCFILE(MYLIB/CSRC) */
/* SRCMBR(GSKCLIENT) */

#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <gskssl.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <errno.h>
#define TRUE 1
#define FALSE 0

void main(void)
{
    gsk_handle my_env_handle=NULL; /* uchwyt środowiska chronionego */
    gsk_handle my_session_handle=NULL; /* uchwyt sesji chronionej */

    struct sockaddr_in address;
    int buf_len, rc = 0, sd = -1;
    int amtWritten, amtRead;
    char buff1[1024];
    char buff2[1024];

    /* Zapisany w kodzie adres IP (należy zmienić na adres serwera) */
    char addr[16] = "1.1.1.1";

    /******
    /* Wszystkie komendy są uruchamiane w pętli */
    /* do/while, dzięki czemu czyszczenie odbywa */
    /* się na końcu. */
    /******
    do
    {
        /* otwiera środowisko gsk */
        rc = errno = 0;
        rc = gsk_environment_open(&my_env_handle);
        if (rc != GSK_OK)
        {
            printf("Niepowodzenie gsk_environment_open() z kodem powrotu = %d kod błędu = %d.\n",
                rc,errno);
            printf("kod powrotu %d oznacza %s\n", rc, gsk_strerror(rc));
            break;
        }

        /* ustawia ID aplikacji */
        rc = errno = 0;
        rc = gsk_attribute_set_buffer(my_env_handle,
            GSK_OS400_APPLICATION_ID,
            "MY_CLIENT_APP",
```

```

13);
if (rc != GSK_OK)
{
    printf("Niepowodzenie gsk_attribute_set_buffer() z kodem powrotu = %d kod błędu = %d.\n",
        rc,errno);
    printf("kod powrotu %d oznacza %s\n", rc, gsk_strerror(rc));
    break;
}

/* ustawia tę stronę jako klienta (domyślnie) */
rc = errno = 0;
rc = gsk_attribute_set_enum(my_env_handle,
                            GSK_SESSION_TYPE,
                            GSK_CLIENT_SESSION);

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

/* Protokoły SSL_V2, SSL_V3 i TLS_V1 są włączone domyślnie. */
/* W tym przykładzie wyłączymy protokół SSL_V2. */
rc = errno = 0;
rc = gsk_attribute_set_enum(my_env_handle,
                            GSK_PROTOCOL_SSLV2,
                            GSK_PROTOCOL_SSLV2_OFF);

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

/* Określa, jakiego pakietu szyfrującego użyć. Domyślnie włączona jest */
/* domyślna lista szyfrowania. W tym przykładzie użyjemy jednego. */
rc = errno = 0;
rc = gsk_attribute_set_buffer(my_env_handle,
                              GSK_V3_CIPHER_SPECS,
                              "05", /* SSL_RSA_WITH_RC4_128_SHA */
                              2);

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

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

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

```



```

    break;
}

/* połączenie z serwerem za pomocą ustawionego numeru portu */
memset((char *) &address, 0, sizeof(address));
address.sin_family = AF_INET;
address.sin_port = 13333;
address.sin_addr.s_addr = inet_addr(addr);
rc = connect(sd, (struct sockaddr *) &address, sizeof(address));
if (rc < 0)
{
    perror("Niepowodzenie connect()");
    break;
}

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

/* powiązanie gniazda z sesją chronioną */
rc=errno=0;
rc = gsk_attribute_set_numeric_value(my_session_handle,
                                    GSK_FD,
                                    sd);

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

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

/* Funkcja memset zeruje bufor szesnastkowo */
memset((char *) buff1, 0, sizeof(buff1));

/* wysłanie komunikatu do serwera w ramach bezpiecznej sesji */
strcpy(buff1,"Test of gsk_secure_soc_write \n\n");

/* wysłanie komunikatu do klienta w ramach bezpiecznej sesji */
buf_len = strlen(buff1);
amtWritten = 0;
rc = gsk_secure_soc_write(my_session_handle, buff1, buf_len, &amtWritten);
if (amtWritten != buf_len)
{
    if (rc != GSK_OK)
    {
        printf("Kod powrotu gsk_secure_soc_write() = %d, numer błędu = %d.\n",rc,errno);
        printf("kod powrotu %d oznacza %s\n", rc, gsk_strerror(rc));
        break;
    }
}

```

```

    }
    else
    {
        printf("Funkcja gsk_secure_soc_write() nie zapisała wszystkich danych.\n");
        break;
    }
}

/* wyświetlenie wyników na ekranie */
printf("Funkcja gsk_secure_soc_write() zapisała bajtów: %d ...\n", amtWritten);
printf("%s\n",buff1);

/* Funkcja memset zeruje bufor szesnastkowo */
memset((char *) buff2, 0x00, sizeof(buff2));

/* odebranie komunikatu od klienta w ramach bezpiecznej sesji */
amtRead = 0;
rc = gsk_secure_soc_read(my_session_handle, buff2, sizeof(buff2), &amtRead);

if (rc != GSK_OK)
{
    printf("Kod powrotu gsk_secure_soc_read() = %d, numer błędu = %d.\n",rc,errno);
    printf("kod powrotu %d oznacza %s\n", rc, gsk_strerror(rc));
    break;
}

/* wyświetlenie wyników na ekranie */
printf("Funkcja gsk_secure_soc_read() odebrała bajtów: %d, oto one:\n",
        amtRead);
printf("%s\n",buff2);

} while(FALSE);

/* wyłączenie obsługi SSL dla gniazda */
if (my_session_handle != NULL)
    gsk_secure_soc_close(&my_session_handle);

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

/* zamknięcie połączenia */
if (sd > -1)
    close(sd);

return;
}

```

Pojęcia pokrewne

“Funkcje API Global Secure ToolKit (GSKit)” na stronie 47

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

Odsyłacze pokrewne

“Przykład: chroniony serwer GSKit z asynchronicznym odbieraniem danych” na stronie 118

Poniższego programu przykładowego można użyć do tworzenia chronionego serwera przy użyciu funkcji API Global Secure ToolKit (GSKit).

“Przykład: chroniony serwer GSKit z asynchronicznym uzgadnianiem” na stronie 127

Funkcja ta umożliwia tworzenie chronionych aplikacji serwera, które mogą obsługiwać żądania w sposób asynchroniczny.

Informacje pokrewne

socket()

close()

connect()

```
gsk_environment_open()
gsk_attribute_set_buffer
gsk_attribute_set_enum()
gsk_environment_init()
gsk_secure_soc_open()
gsk_attribute_set_numeric_value()
gsk_secure_soc_init()
gsk_secure_soc_close()
gsk_environment_close()
gsk_secure_soc_write ()
gsk_secure_soc_startInit()
gsk_secure_soc_startRecv()
gsk_secure_soc_read ()
```

Przykład: chroniony serwer używający funkcji API SSL_

Tworząc aplikacje chronione można oprócz funkcji API GSKit używać funkcji API SSL_. Funkcje API SSL_ APIs występują tylko w systemie operacyjnym i5/OS.

Poniższa ilustracja przedstawia funkcje API gniazd i SSL_ używane do tworzenia serwera chronionego.

Przebieg zdarzeń w gnieździe: chroniony serwer używający funkcji API SSL_

Poniżej opisano relacje pomiędzy funkcjami API umożliwiającymi pracę serwera SSL i jego komunikację z klientem SSL:

1. Wywołanie funkcji SSL_Init() lub SSL_Init_Application() w celu zainicjowania środowiska pracy dla przetwarzania SSL i w celu określenia informacji o ochronie SSL dla wszystkich sesji SSL, które będą uruchamiane w bieżącym zadaniu. Należy użyć tylko jednej z tych funkcji API. Zalecane jest użycie funkcji API SSL_Init_Application().

Uwaga: Poniższy program przykładowy używa funkcji API SSL_Init_Application.

2. Serwer wywołuje funkcję socket() w celu uzyskania deskryptora gniazda.
3. Serwer wywołuje funkcje bind(), listen() i accept(), aby uaktywnić połączenie dla programu serwera.
4. Serwer wywołuje funkcję SSL_Create(), aby włączyć obsługę SSL dla podłączonego gniazda.
5. Serwer wywołuje funkcję SSL_Handshake(), aby zainicjować uzgadnianie SSL parametrów szyfrujących.
6. Serwer wywołuje funkcje SSL_Write() i SSL_Read(), aby wysłać i odebrać dane.
7. Serwer wywołuje funkcję SSL_Destroy() w celu wyłączenia obsługi SSL dla gniazda.
8. Serwer wywołuje funkcję close() w celu usunięcia podłączonych gniazd.

Przebieg zdarzeń w gnieździe: chroniony klient używający funkcji API SSL_

1. Wywołanie funkcji SSL_Init() lub SSL_Init_Application() w celu zainicjowania środowiska pracy dla przetwarzania SSL i w celu określenia informacji o ochronie SSL dla wszystkich sesji SSL, które będą uruchamiane w bieżącym zadaniu. Należy użyć tylko jednej z tych funkcji API. Zalecane jest użycie funkcji API SSL_Init_Application().

Uwaga: Poniższy program przykładowy używa funkcji API SSL_Init_Application.

2. Klient wywołuje funkcję socket() w celu uzyskania deskryptora gniazda.
3. Klient wywołuje funkcję connect() w celu uaktywnienia połączenia dla programu klienta.
4. Klient wywołuje funkcję SSL_Create(), aby włączyć obsługę SSL dla podłączonego gniazda.

5. Klient wywołuje funkcję SSL_Handshake(), aby zainicjować uzgadnianie SSL parametrów szyfrujących.
6. Klient wywołuje funkcje SSL_Read() i SSL_Write() w celu odebrania i wysłania danych.
7. Klient wywołuje funkcję SSL_Destroy() w celu wyłączenia obsługi SSL dla gniazda.
8. Klient wywołuje funkcję close() w celu usunięcia podłączonych gniazd.

Uwaga: W przykładzie użyto rodziny adresów AF_INET; można go jednak zmodyfikować, aby została użyta rodzina adresów AF_INET6. Korzystając z przykładowego kodu, użytkownik akceptuje warunki określone w temacie “Informacje dotyczące licencji na kod” na stronie 192.

```

/* Program serwera SSL używający funkcji SSL_Init_Application */

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

/* Użyj następującej komendy, aby utworzyć program */
/* skonsolidowany:                                 */
/* CRTBNDC PGM(MYLIB/SSLSERVAPP)                   */
/*          SRCFILE(MYLIB/CSRC)                     */
/*          SRCMBR(SSLSERVAPP)                       */

#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <qsoss1.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 */
lfd = socket(AF_INET, SOCK_STREAM, 0);
if (lfd < 0)
{
    perror("Niepowodzenie socket()");
    return;
}

/* ustawienie gniazda do natychmiastowego ponownego użycia */
rc = setsockopt(lfd, SOL_SOCKET,
                SO_REUSEADDR,
                (char *)&on,
                sizeof(on));

if (rc < 0)
{
    perror("Niepowodzenie setsockopt()");
    return;
}

/* powiązanie do adresu lokalnego serwera */
memset((char *) &address, 0, sizeof(address));
address.sin_family = AF_INET;
address.sin_port = 13333;
address.sin_addr.s_addr = 0;
rc = bind(lfd, (struct sockaddr *) &address, sizeof(address));
if (rc < 0)
{
    perror("Niepowodzenie bind()");
    close(lfd);
    return;
}

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

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

```

```

/* ustawienie parametrów dla uzgadniania */
sslh -> protocol = 0;
sslh -> timeout = 0;
sslh -> cipherSuiteList = &cipher;
sslh -> cipherSuiteListLen = 1;

/* inicjowanie uzgodnienia SSL */
rc = SSL_Handshake(sslh, SSL_HANDSHAKE_AS_SERVER);
if (rc != 0)
{
    printf("Niepowodzenie SSL_Handshake() z kodem powrotu = %d i kodem błędu = %d.\n",
        rc,errno);
    SSL_Destroy(sslh);
    close(lsd);
    close(sd);
    return;
}

/* Funkcja memset zeruje bufor szesnastkowo */
memset((char *) buff, 0, sizeof(buff));

/* odebranie komunikatu od klienta w ramach bezpiecznej sesji */
rc = SSL_Read(sslh, buff, sizeof(buff));
if (rc < 0)
{
    printf("SSL_Read() rc = %d i errno = %d.\n",rc,errno);
    rc = SSL_Destroy(sslh);
    if (rc != 0)
        printf("SSL_Destroy() rc = %d i errno = %d.\n",rc,errno);
    close(lsd);
    close(sd);
    return;
}

/* wyświetlenie wyników na ekranie */
printf("SSL_Read() odczytała ...\n");
printf("%s\n",buff);

/* wysłanie komunikatu do klienta w ramach bezpiecznej sesji */
buf_len = strlen(buff);
rc = SSL_Write(sslh, buff, buf_len);
if (rc != buf_len)
{
    if (rc < 0)
    {
        printf("Niepowodzenie SSL_Write() z rc = %d.\n",rc);
        SSL_Destroy(sslh);
        close(lsd);
        close(sd);
        return;
    }
    else
    {
        printf("SSL_Write() nie zapisała wszystkich danych.\n");
        SSL_Destroy(sslh);
        close(lsd);
        close(sd);
        return;
    }
}

/* wyświetlenie wyników na ekranie */
printf("SSL_Write() zapisała ...\n");
printf("%s\n",buff);

/* wyłączenie obsługi SSL dla gniazda */

```

```

SSL_Destroy(sslh);

/* zamknięcie połączenia */
close(sd);

/* zamknięcie gniazda nasłuchującego */
close(lsd);

return;
}

```

Pojęcia pokrewne

“Funkcje API SSL_” na stronie 50

Funkcje API SSL_ umożliwiają tworzenie aplikacji gniazd chronionych w systemie iSeries.

Odsyłacze pokrewne

“Przykład: chroniony klient używający funkcji API SSL_”

Poniższy przykład to aplikacja klienta używająca funkcji API SSL_ do komunikowania się z aplikacją serwera używającą funkcji API SSL_:

Informacje pokrewne

```

SSL_Init()
SSL_Init_Application()
socket()
listen()
bind()
accept()
close()
connect()
SSL_Create()
SSL_Destroy()
SSL_Handshake()
SSL_Read()
SSL_Write()

```

Przykład: chroniony klient używający funkcji API SSL_

Poniższy przykład to aplikacja klienta używająca funkcji API SSL_ do komunikowania się z aplikacją serwera używającą funkcji API SSL_:

Uwaga: Korzystając z przykładowego kodu, użytkownik akceptuje warunki określone w temacie “Informacje dotyczące licencji na kod” na stronie 192.

```

/* Program klienta SSL używający funkcji SSL_Init_Application */

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

/* Użyj następującej komendy, aby utworzyć program */
/* skonsolidowany:                                 */
/* CRTBNDC PGM(MYLIB/SSLCLIAPP)                     */
/*          SRCFILE(MYLIB/CSRC)                      */
/*          SRCMBR(SSLCLIAPP)                        */

#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <ctype.h>

```

```

#include <sys/socket.h>
#include <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[12] = "16.35.146.84";

    void * malloc_ptr = (void *) NULL;
    unsigned int malloc_size = 8192;

    /* memset - szesnastkowe zerowanie struktury sslinit */
    memset((char *)&sslinit, 0, sizeof(sslinit));

    /* wypełnianie wartościami struktury sslinit */
    /* z użyciem istniejącego ID aplikacji */
    sslinit.applicationID = "MY_CLIENT_APP";
    sslinit.applicationIDLen = 13;
    sslinit.localCertificate = NULL;
    sslinit.localCertificateLen = 0;
    sslinit.cipherSuiteList = NULL;
    sslinit.cipherSuiteListLen = 0;

    /* przydzielanie i ustawianie wskaźników dla buforu certyfikatu */
    malloc_ptr = (void*) malloc(malloc_size);
    sslinit.localCertificate = (unsigned char*) malloc_ptr;
    sslinit.localCertificateLen = malloc_size;

    /* inicjowanie wywołania SSL SSL_Init_Application */
    rc = SSL_Init_Application(&sslinit);
    if (rc != 0)
    {
        printf("Niepowodzenie SSL_Init_Application() z kodem powrotu = %d i kodem błędu = %d.\n",
            rc,errno);
        return;
    }

    /* inicjowanie gniazda */
    sd = socket(AF_INET, SOCK_STREAM, 0);
    if (sd < 0)
    {
        perror("Niepowodzenie socket()");
        return;
    }

    /* uaktywnienie obsługi SSL dla gniazda */
    sslh = SSL_Create(sd, SSL_ENCRYPT);
    if (sslh == NULL)
    {
        printf("Niepowodzenie SSL_Create() z errno = %d.\n", errno);
        close(sd);
        return;
    }
}

```



```

}

/* połączenie z serwerem za pomocą ustawionego numeru portu */
memset((char *) &address, 0, sizeof(address));
address.sin_family = AF_INET;
address.sin_port = 13333;
address.sin_addr.s_addr = inet_addr(addr);
rc = connect(sd, (struct sockaddr *) &address, sizeof(address));
if (rc < 0)
{
    perror("Niepowodzenie connect()");
    close(sd);
    return;
}

/* przygotowanie do wywołania uzgodnienia, ustawianie algorytmu */
sslh -> protocol = 0;
sslh -> timeout = 0;
sslh -> cipherSuiteList = &cipher;
sslh -> cipherSuiteListLen = 1;

/* inicjowanie uzgodnienia SSL - jako KLIENT */
rc = SSL_Handshake(sslh, SSL_HANDSHAKE_AS_CLIENT);
if (rc != 0)
{
    printf("Niepowodzenie SSL_Handshake() z kodem powrotu = %d i kodem błędu = %d.\n",
        rc,errno);
    close(sd);
    return;
}

/* wysłanie komunikatu do serwera w ramach bezpiecznej sesji */
strcpy(buff1,"Test funkcji SSL_Write \n\n");
buf_len = strlen(buff1);
rc = SSL_Write(sslh, buff1, buf_len);
if (rc != buf_len)
{
    if (rc < 0)
    {
        printf("Niepowodzenie SSL_Write() z rc = %d i errno = %d.\n",rc, errno);
        SSL_Destroy(sslh);
        close(sd);
        return;
    }
    else
    {
        printf("SSL_Write() nie zapisała wszystkich danych.\n");
        SSL_Destroy(sslh);
        close(sd);
        return;
    }
}

/* wyświetlenie wyników na ekranie */
printf("SSL_Write() zapisała ...\n");
printf("%s\n",buff1);

memset((char *) buff2, 0x00, sizeof(buff2));

/* odebranie komunikatu z serwera w ramach bezpiecznej sesji */
rc = SSL_Read(sslh, buff2, buf_len);
if (rc < 0)
{
    printf("Niepowodzenie SSL_Read() z rc = %d.\n",rc);
    SSL_Destroy(sslh);
    close(sd);
    return;
}

```

```

}

/* wyświetlenie wyników na ekranie */
printf("SSL_Read() odczytała ...\n");
printf("%s\n",buff2);

/* wyłączenie obsługi SSL dla gniazda */
SSL_Destroy(sslh);

/* zamknięcie połączenia przez zamknięcie lokalnego gniazda */
close(sd);
return;
}

```

Pojęcia pokrewne

“Funkcje API SSL_” na stronie 50

Funkcje API SSL_ umożliwiają tworzenie aplikacji gniazd chronionych w systemie iSeries.

Odsyłacze pokrewne

“Przykład: chroniony serwer używający funkcji API SSL_” na stronie 141

Tworząc aplikacje chronione można oprócz funkcji API GSKit używać funkcji API SSL_. Funkcje API SSL_ APIs występują tylko w systemie operacyjnym i5/OS.

Przykład: procedury sieciowe obsługujące ochronę wątków i używające funkcji gethostbyaddr_r()

Poniżej przedstawiono przykład programu, w którym wykorzystano funkcję gethostbyaddr_r().

Wszystkie pozostałe procedury, które używają nazw zakończonych na "_r", mają podobną semantykę i także obsługują chronione wątki. Ten przykładowy program pobiera adres IP w notacji dziesiętnej z kropkami i drukuje nazwę hosta.

Uwaga: Korzystając z przykładowego kodu, użytkownik akceptuje warunki określone w temacie “Informacje dotyczące licencji na kod” na stronie 192.

```

/*****
/* 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 NUPARMS 2
/*****
/* Przekaż parametr, który jest adresem IP w notacji */
/* dziesiętnej z kropkami. Nazwa hosta zostanie */
/* wyświetlona, jeśli będzie znaleziona; w przeciwnym */
/* razie wyświetl komunikat 'hosta nie znaleziono'. */
*****/
int main (int argc, char *argv[])
{
    int rc;
    struct in_addr internet_address;
    struct hostent hst_ent;
    struct hostent_data hst_ent_data;
    char dotted_decimal_address [16];
    char host_name[MAXHOSTNAMELEN];

    /*****
    /* Sprawdź liczbę przekazanych argumentów */
    *****/
    if (argc != NUPARMS)

```

```

{
    printf("Zła liczba wpisanych argumentów parametrów\n");
    exit(-1);
}
/*****
/* Uzyskaj adresowalność przekazanych parametrów */
*****/
strcpy(dotted_decimal_address, argv[1]);

/*****
/* Zainicjuj pole struktury */
/* hostent_data.host_control_blk szesnastkowymi zerami */
/* zanim zostanie użyte. Jeśli wymagasz zgodności z innymi*/
/* platformami, musisz zainicjować całą strukturę */
/* hostent_data szesnastkowymi zerami. */
*****/
/* Zainicjuj strukturę hostent_data do szesnastkowych 00 */
*****/
memset(&hst_ent_data, HEX00, sizeof(struct hostent_data));

/*****
/* Przetłumacz adres internetowy z postaci dziesiętnej */
/* z kropkami do formatu 32-bitowego adresu IP. */
*****/
internet_address.s_addr=inet_addr(dotted_decimal_address);

/*****
/* Uzyskaj nazwę hosta */
*****/
/*****
/* UWAGA: gethostbyaddr_r() zwraca liczbę całkowitą. */
/* Oto możliwe wartości: */
/* -1 (wywołanie nie powiodło się) */
/* 0 (wywołanie powiodło się) */
*****/
rc=gethostbyaddr_r((char *) &internet_address,
                  sizeof(struct in_addr), AF_INET,
                  &hst_ent, &hst_ent_data);

if (rc == -1)
{
    printf("Nie znaleziono nazwy hosta\n");
    exit(-1);
}
else
{
/*****
/* Skopiuj nazwę hosta do buforu danych wyjściowych*/
*****/
(void) memcpy((void *) host_name,
/*****
/* Wszystkie rezultaty należy adresować poprzez */
/* strukturę hostent (hst_ent). */
/* UWAGA: Struktura danych hostent_data */
/* (hst_ent_data) jest tylko repozytorium */
/* danych używanym do obsługi struktury */
/* hostent. Aplikacje powinny traktować strukturę */
/* hostent_data jako obszar przechowywania danych */
/* poziomu hosta, do których */
/* nie muszą mieć dostępu. */
*****/
                (void *) hst_ent.h_name,
                MAXHOSTNAMELEN);
/*****
/* Wydrukuj nazwę hosta */
*****/
printf("Nazwa hosta to %s\n", host_name);
}
}

```

```
    }  
    exit(0);  
}
```

Pojęcia pokrewne

“Ochrona wątków” na stronie 57

Funkcja realizuje ochronę wątków, jeśli można ją uruchomić jednocześnie w wielu wątkach w ramach tego samego procesu.

Odsyłacze pokrewne

“Funkcje sieciowe gniazd” na stronie 63

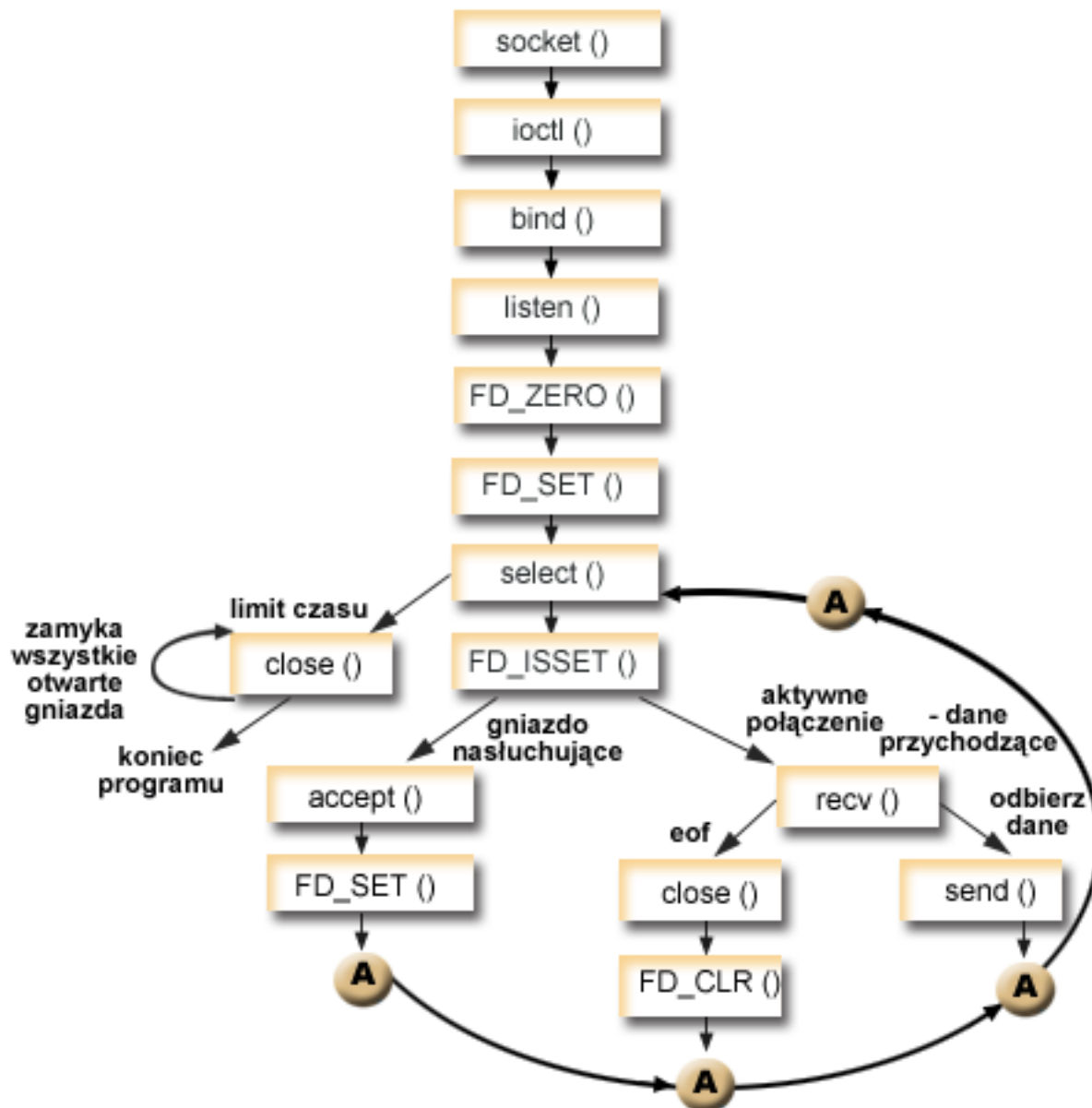
Funkcje sieciowe gniazd umożliwiają aplikacjom uzyskiwanie informacji od hostów, protokołów, usług oraz plików sieciowych.

Informacje pokrewne

gethostbyaddr_r()

Przykład: nieblokujące operacje we/wy i funkcja select()

Poniższy program przykładowy 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 `socket()` zwraca deskryptor gniazda odpowiadający punktowi końcowemu. Instrukcja ta informuje również, że dla tego gniazda zostanie użyta rodzina adresów INET (Internet Protocol) z transportem TCP transport (`SOCK_STREAM`).
2. Funkcja `ioctl()` umożliwi ponowne użycie adresu lokalnego po restarcie serwera, zanim upłynie wymagany czas oczekiwania. W tym przykładzie, ustawia gniazdo na nieblokujące. Wszystkie gniazda dla połączeń przychodzących również są nieblokujące, ponieważ będą dziedziczyć ten stan od gniazda nasłuchującego.
3. Po utworzeniu deskryptora gniazda funkcja `bind()` pobiera unikalną nazwę gniazda.

4. Funkcja listen() umożliwia serwerowi przyjęcie połączenia przychodzącego od klienta.
5. Serwer używa funkcji accept() do zaakceptowania żądania połączenia przychodzącego. Wywołanie funkcji accept() zostanie zablokowane na nieokreślony czas oczekiwania na połączenie przychodzące.
6. Funkcja select() powoduje, że proces oczekuje na wystąpienie zdarzenia, po którym kontynuuje działanie. W tym przykładzie funkcja select () zwraca liczbę odpowiadającą deskryptorom gniazda, które są gotowe do przetwarzania.
 - 0 Wskazuje, że następuje przekroczenie limitu czasu procesu. W tym przykładzie limit czasu ustawiono na 30 sekund.
 - 1 Wskazuje, że proces nie powiódł się.
 - 1 Wskazuje, że tylko jeden deskryptor jest gotowy do przetwarzania. W tym przykładzie zwrócenie wartości 1 powoduje, że FD_ISSET i kolejne wywołania gniazd zostaną zakończone tylko raz.
 - n Wskazuje, że na przetwarzanie czeka wiele deskryptorów. W tym przykładzie zwrócenie wartości n powoduje, że FD_ISSET i dalszy kod zapętłają się i kończą obsługę żądań w kolejności ich odebrania przez serwer.
7. Funkcje accept() i recv() kończą działanie po zwróceniu wartości EWOULDBLOCK.
8. Funkcja send() odsyła dane do klienta.
9. Funkcja close() zamyka wszystkie otwarte deskryptory gniazd.

Uwaga: Korzystając z przykładowego kodu, użytkownik akceptuje warunki określone w temacie “Informacje dotyczące licencji na kod” na stronie 192.

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/ioctl.h>
#include <sys/socket.h>
#include <sys/time.h>
#include <netinet/in.h>
#include <errno.h>

#define SERVER_PORT 12345

#define TRUE      1
#define FALSE    0

main (int argc, char *argv[])
{
    int    i, len, rc, on = 1;
    int    listen_sd, max_sd, new_sd;
    int    desc_ready, end_server = FALSE;
    int    close_conn;
    char   buffer[80];
    struct sockaddr_in  addr;
    struct timeval      timeout;
    struct fd_set       master_set, working_set;

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

    /******
    /* Umożliwia ponowne użycie deskryptora gniazda */
    /******
    rc = setsockopt(listen_sd, SOL_SOCKET, SO_REUSEADDR,
```

```

        (char *)&on, sizeof(on));
if (rc < 0)
{
    perror("Niepowodzenie setsockopt()");
    close(listen_sd);
    exit(-1);
}

/*****
/* Ustawia 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 ioctl()");
    close(listen_sd);
    exit(-1);
}

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

/*****
/* Ustawia nasłuchiwanie parametru backlog.
*****/
rc = listen(listen_sd, 32);
if (rc < 0)
{
    perror("Niepowodzenie listen()");
    close(listen_sd);
    exit(-1);
}

/*****
/* Inicjuje główny fd_set
*****/
FD_ZERO(&master_set);
max_sd = listen_sd;
FD_SET(listen_sd, &master_set);

/*****
/* Inicjuje strukturę timeval wartością 3 min. Brak
/* aktywności w tym czasie spowoduje zakończenie programu.
*****/
timeout.tv_sec = 3 * 60;
timeout.tv_usec = 0;

/*****
/* Pętla oczekiwania na przychodzące połączenia lub dane
/* dla dowolnego połączonego gniazda.
*****/
do

```

```

{
/*****/
/* Kopiuje główny fd_set do roboczego fd_set. */
/*****/
memcpy(&working_set, &master_set, sizeof(master_set));

/*****/
/* Wywołuje select() i czeka 5 min na zakończenie. */
/*****/
printf("Oczekiwanie na select()...\n");
rc = select(max_sd + 1, &working_set, NULL, NULL, &timeout);

/*****/
/* Sprawdzenie, czy wywołanie funkcji select się powiodło.*/
/*****/
if (rc < 0)
{
    perror("Niepowodzenie select()");
    break;
}

/*****/
/* Sprawdzenie, czy nie upłynął czas oczekiwania 5 min. */
/*****/
if (rc == 0)
{
    printf("Przekroczenie czasu dla select(). Koniec programu.\n");
break;
}

/*****/
/* Przynajmniej jeden deskryptor jest czytelny. Trzeba */
/* sprawdzić, który z dekskryptorów. */
/*****/
desc_ready = rc;
for (i=0; i <= max_sd && desc_ready > 0; ++i)
{
/*****/
/* Sprawdzenie, czy deskryptor jest gotowy. */
/*****/
if (FD_ISSET(i, &working_set))
{
/*****/
/* Znaleziono czytelny deskryptor - o jeden mniej */
/* do znalezienia. Czynność tę powtarza się, więc */
/* po znalezieniu wszystkich gotowych deskryptorów */
/* można zakończyć przeszukiwanie zestawu */
/* roboczego. */
/*****/
desc_ready -= 1;

/*****/
/* Sprawdzenie, czy to nasłuchujące gniazdo. */
/*****/
if (i == listen_sd)
{
    printf("Gniazdo nasłuchujące jest gotowe\n");
/*****/
/* Akceptowanie wszystkich połączeń przychodz. */
/* które znajdują się w kolejce gniazda */
/* nasłuchującego przed ponownym wywołaniem. */
/*****/
do
{
/*****/
/* Akceptowanie każdego połączenia przychodz. */
/* Jeśli akceptowanie EWOULDBLOCK nie

```



```

/* powiedzie się, to znaczy, że */
/* 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 accept()");
        end_server = TRUE;
    }
    break;
}

/*****/
/* Dodanie nowego połączenia przychodzącego */
/* do głównego zestawu operacji odczytu. */
/*****/
printf("Nowe połączenie przychodzące - %d\n", new_sd);
FD_SET(new_sd, &master_set);
if (new_sd > max_sd)
    max_sd = new_sd;

/*****/
/* Powrót w pętli i akceptowanie nowego */
/* połączenia przychodzącego. */
/*****/
} while (new_sd != -1);
}

/*****/
/* To nie jest gniazdo nasłuchujące, dlatego */
/* istniejące połączenie musi być czytelne */
/*****/
else
{
    printf("Deskryptor %d jest czytelny\n", i);
    close_conn = FALSE;
    /*****/
    /* Odbiór wszystkich danych przychodzących do */
    /* tego gniazda przed powrotem w pętli i ponownym*/
    /* wywołaniem funkcji select. */
    /*****/
    do
    {
        /*****/
        /* Odbieranie danych dla tego połączenia, */
        /* dopóki recv nie powiedzie się z wartością */
        /* EWOULDBLOCK. Jeśli wystąpi inne */
        /* niepowodzenie, połączenie zost. zamknięte. */
        /*****/
        rc = recv(i, buffer, sizeof(buffer), 0);
        if (rc < 0)
        {
            if (errno != EWOULDBLOCK)
            {
                perror("Niepowodzenie recv()");
                close_conn = TRUE;
            }
            break;
        }
    }

    /*****/
    /* Sprawdzenie, czy połączenie nie zostało */
    /* zamknięte przez klienta. */

```

```

/*****/
if (rc == 0)
{
    printf(" Połączenie zamknięte\n");
    close_conn = TRUE;
    break;
}

/*****/
/* Dane zostały odebrane. */
/*****/
len = rc;
printf("Otrzymano bajtów: %d\n", len);

/*****/
/* Odesłanie danych do klienta */
/*****/
rc = send(i, buffer, len, 0);
if (rc < 0)
{
    perror("Niepowodzenie send()");
    close_conn = TRUE;
    break;
}

} while (TRUE);

/*****/
/* Jeśli opcja close_conn została włączona, */
/* trzeba wyczyścić aktywne połączenie. Procedura */
/* czyszcząca obejmuje usunięcie deskryptora z */
/* zestawu głównego i określenie nowej wartości */
/* maksymalnej deskryptora na podstawie bitów, */
/* które 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;
    }
}
} /* Koniec "istniejące połączenie musi być czytelne" */
} /* Koniec "if (FD_ISSET(i, &working_set))" */
} /* Koniec pętli poprzez wybierane deskryptory */

} while (end_server == FALSE);

/*****/
/* Czyszczenie wszystkich otartych gniazd */
/*****/
for (i=0; i <= max_sd; ++i)
{
    if (FD_ISSET(i, &master_set))
        close(i);
}
}

```

Pojęcia pokrewne

“Nieblokujące operacje we/wy” na stronie 57

Gdy aplikacja wywoła jedną z funkcji wejściowych gniazda, a nie będzie żadnych danych do odczytania, funkcja zablokuje się i nie zakończy się, dopóki nie danych do odczytania.

“Multipleksowanie operacji we/wy - funkcja select()” na stronie 63

Asynchroniczne operacje we/wy zapewniają znacznie efektywniejszą metodę maksymalizowania zasobów aplikacji niż funkcja API select(), dlatego zalecane jest używanie tych operacji. Jednak konkretny projekt aplikacji może dopuszczać użycie funkcji select().

Odsyłacze pokrewne

“Zalecenia dotyczące projektowania aplikacji używających gniazd” na stronie 83

Aby projektować aplikacje używające gniazd, należy określić wymagania funkcjonalne, cele i potrzeby tej aplikacji.

“Przykład: ogólny program klienta” na stronie 108

Poniższy kod zawiera przykłady podstawowych zadań klienta.

Informacje pokrewne

accept()

recv()

ioctl()

send()

listen()

close()

socket()

bind()

select()

Używanie funkcji poll() zamiast funkcji select()

Funkcja poll() jest częścią Pojedynczej specyfikacji Unix i standardu UNIX 95/98. Funkcja poll() spełnia tę samą funkcję co funkcja select(). Jedyną różnicą pomiędzy nimi jest interfejs programu wywołującego.

Funkcja select() wymaga, by aplikacja przesyłała dane jako szereg bitów, gdzie każdy bit oznacza numer deskryptora. Jeśli numery deskryptorów są bardzo duże, może to spowodować przepełnienie przydzielonych 30KB pamięci, powodując tym samym wielokrotne powtórzenia tego procesu. Może to obniżyć wydajność.

Funkcja poll() umożliwia aplikacji wysłanie szeregu struktur raczej niż 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 socket() zwraca deskryptor gniazda odpowiadający punktowi końcowemu. Instrukcja ta informuje również, że dla tego gniazda zostanie użyta rodzina adresów INET (Internet Protocol) z transportem TCP transport (SOCK_STREAM).
2. Funkcja setsockopt umożliwia aplikacji ponowne użycie adresu lokalnego po restarcie serwera, zanim upłynie wymagany czas oczekiwania.
3. Funkcja ioctl ustawia gniazdo na nieblokujące. Wszystkie gniazda dla połączeń przychodzących również są nieblokujące, ponieważ będą dziedziczyć ten stan od gniazda nasłuchującego.
4. Po utworzeniu deskryptora gniazda funkcja bind() pobiera unikalną nazwę gniazda.
5. Funkcja listen() umożliwia serwerowi przyjęcie połączenia przychodzącego od klienta.
6. Funkcja select() powoduje, że proces oczekuje na wystąpienie zdarzenia, po którym kontynuuje działanie. Funkcja może zwrócić następujące wartości łańcucha.

0 Wskazuje, że następuje przekroczenie limitu czasu procesu. W tym przykładzie limit czasu ustawiono na 3 minuty (w milisekundach).

-1 Wskazuje, że proces nie powiódł się.

- 1 Wskazuje, że tylko jeden deskryptor jest gotowy do przetwarzania, i że jest przetwarzany tylko jeśli gniazdem nasłuchującym.
 - 1++ Wskazuje, że na przetwarzanie czeka wiele deskryptorów. Funkcja poll() umożliwi symultaniczne połączenie ze wszystkimi deskryptorami w kolejce gniazda nasłuchującego.
7. Funkcje accept() i recv() kończą działanie po zwróceniu wartości EWOULDBLOCK.
 8. Funkcja send() odsyła dane do klienta.
 9. Funkcja close() zamyka wszystkie otwarte deskryptory gniazd.

Uwaga: Korzystając z przykładowego kodu, użytkownik akceptuje warunki określone w temacie “Informacje dotyczące licencji na kod” na stronie 192.

```
#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;

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

    /******
    /* Umożliwia ponowne użycie deskryptora gniazda */
    /******
    rc = setsockopt(listen_sd, SOL_SOCKET, SO_REUSEADDR,
                    (char *)&on, sizeof(on));

    if (rc < 0)
    {
        perror("Niepowodzenie setsockopt()");
        close(listen_sd);
        exit(-1);
    }

    /******
    /* Ustawia 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 *)&n);
if (rc < 0)
{
    perror("Niepowodzenie ioctl()");
    close(listen_sd);
    exit(-1);
}

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

/*****/
/* Ustawia nasłuchiwanie parametru backlog. */
/*****/
rc = listen(listen_sd, 32);
if (rc < 0)
{
    perror("Niepowodzenie listen()");
    close(listen_sd);
    exit(-1);
}

/*****/
/* Inicjowanie struktury pollfd */
/*****/
memset(fds, 0, sizeof(fds));

/*****/
/* Konfiguracja początkowego gniazda nasłuchującego */
/*****/
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 liczony jest w milisekundach. */
/*****/
timeout = (3 * 60 * 1000);

/*****/
/* Pętla oczekiwania na przychodzące połączenia lub dane */
/* dla dowolnego połączonego gniazda. */
/*****/
do
{
    /*****/
    /* Wywołuje select() i czeka 3 min na zakończenie. */
    /*****/
    printf("Oczekiwanie na select()...\n");
    rc = poll(fds, nfd, timeout);

    /*****/
    /* Sprawdzenie, czy wywołanie poll select się powiodło.*/

```

```

/*****/
if (rc < 0)
{
    perror("Niepowodzenie poll()");
    break;
}

/*****/
/* Sprawdzenie, czy nie upłynął czas oczekiwania 3 min . */
/*****/
if (rc == 0)
{
    printf("Przekroczenie limitu czasu poll(). Koniec programu.\n");
    break;
}

/*****/
/* Przynajmniej jeden deskryptor jest czytelny. Trzeba */
/* sprawdzić, który. */
/*****/
current_size = nfds;
for (i = 0; i < current_size; i++)
{
    /*****/
    /* Znajdź deskryptory, które zwróciły */
    /* wartość POLLIN, i określ czy jest to nasłuchujące */
    /* czy aktywne połączenie. */
    /*****/
    if(fds[i].revents == 0)
        continue;

    /*****/
    /* Jeśli pole revents nie ma wartości POLLIN, jest */
    /* to nieoczekiwany rezultat, należy sporządzić */
    /* prot. 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)
    {
        /*****/
        /* Nasłuchujący deskryptor jest czytelny. */
        /*****/
        printf("Gniazdo nasłuchujące jest gotowe\n");

        /*****/
        /* Akceptowanie wszystkich połączeń przychodz., */
        /* które znajdują się w kolejce gniazda */
        /* nasłuchującego przed ponownym wywołaniem */
        /* funkcji poll. */
        /*****/
        do
        {
            /*****/
            /* Akceptowanie każdego połączenia przychodz. */
            /* Jeśli akceptowanie nie powiedzie się z */
            /* wartością EWOULDBLOCK, to znaczy, że */
            /* 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 accept()");
        end_server = TRUE;
    }
    break;
}

/*****
/* Dodanie nowego połączenia przychodzącego      */
/* dla struktury pollfd                            */
*****/
printf("Nowe połączenie przychodzące - %d\n", new_sd);
fds[nfds].fd = new_sd;
fds[nfds].events = POLLIN;
nfds++;

/*****
/* Powrót w pętli i akceptowanie nowego          */
/* połączenia przychodzącego.                    */
*****/
} while (new_sd != -1);
}

/*****
/* To nie jest gniazdo nasłuchujące, dlatego      */
/* istniejące połączenie musi być czytelne       */
*****/

else
{
    printf("Deskryptor %d jest czytelny\n", i);
    close_conn = FALSE;
    /*****
    /* Odbiór wszystkich danych przychodzących do  */
    /* tego gniazda przed powrotem w pętli i ponownym */
    /* wywołaniem funkcji poll.                      */
    *****/

    do
    {
        /*****
        /* Odbieranie danych dla tego połączenia,    */
        /* dopóki recv nie powie o sobie z wartością  */
        /* EWOULDBLOCK. Jeśli wystąpi inne           */
        /* niepowodzenie, połączenie zost. zamknięte. */
        *****/
        rc = recv(fds[i].fd, buffer, sizeof(buffer), 0);
        if (rc < 0)
        {
            if (errno != EWOULDBLOCK)
            {
                perror("Niepowodzenie recv()");
                close_conn = TRUE;
            }
            break;
        }
    }

    /*****
    /* Sprawdzenie, czy połączenie nie zostało     */
    /* zamknięte przez klienta.                    */
    *****/
    if (rc == 0)
    {

```

```

    printf(" Połączenie zamknięte\n");
    close_conn = TRUE;
    break;
}

/*****
/* Dane zostały odebrane. */
*****/
len = rc;
printf("Otrzymano bajtów: %d\n", len);

/*****
/* Odesłanie danych do klienta */
*****/
rc = send(fds[i].fd, buffer, len, 0);
if (rc < 0)
{
    perror("Niepowodzenie send()");
    close_conn = TRUE;
    break;
}

} while (TRUE);

/*****
/* Jeśli opcja close_conn została włączona, */
/* trzeba wyczyścić aktywne połączenie. Procedura */
/* czyszcząca obejmuje usunięcie deskryptora. */
/* */
*****/
if (close_conn)
{
    close(fds[i].fd);
    fds[i].fd = -1;
    compress_array = TRUE;
}

} /* Koniec "istniejące połączenie musi być czytelne" */
} /* Koniec pętli poprzez deskryptory do odpytania */

/*****
/* Jeśli opcja compress array została włączona, należy */
/* skompresować tablice i zmniejszyć liczbę deskryptorów */
/* zbiorów. Nie jest konieczne 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. */

```



```

/*****
/* Czyszczenie wszystkich otartych gniazd          */
/*****
for (i = 0; i < nfd; i++)
{
    if(fds[i].fd >= 0)
        close(fds[i].fd);
}
}

```

Informacje pokrewne

accept()
recv()
ioctl()
send()
listen()
close()
socket()
bind()
setsockopt
poll()

Przykład: używanie sygnałów z blokującymi funkcjami API gniazd

Sygnały powiadamiają o zablokowaniu procesu lub aplikacji.

Udostępniają one limit czasu blokowania procesów. W tym przykładzie sygnał pojawia się po pięciu sekundach po wywołaniu funkcji accept(). Wywołanie to zwykle blokuje proces na czas nieokreślony, lecz ponieważ ustawiony został alarm, wywołanie zostanie zablokowane jedynie na czas pięciu sekund. Ponieważ zablokowane programy mogą pogarszać wydajność aplikacji lub serwera, można użyć sygnałów do zmniejszenia tego wpływu. Poniższy przykład pokazuje, w jaki sposób używać sygnałów z blokującymi funkcjami API gniazd.

Uwaga: Zamiast konwencjonalnego modelu preferowane jest użycie asynchronicznych operacji we/wy w modelu serwera z obsługą wątków.



Przebieg zdarzeń w gnieździe: używanie sygnałów z blokowaniem gniazd

Opisane poniżej wywołania funkcji ilustrują sposób użycia sygnałów do powiadomienia aplikacji o tym, że gniazdo jest nieaktywne.

1. Funkcja `socket()` zwraca deskryptor gniazda odpowiadający punktowi końcowemu. Instrukcja ta informuje również, że dla tego gniazda zostanie użyta rodzina adresów INET (Internet Protocol) z transportem TCP transport (`SOCK_STREAM`).
2. Po utworzeniu deskryptora gniazda funkcja `bind()` pobiera unikalną nazwę gniazda. W tym przykładzie numer portu nie jest określony, ponieważ aplikacja kliencka nie nawiązuje połączenia z gniazdem. Tego fragmentu kodu można użyć w innych serwerach, które korzystają z blokowania takich funkcji API, jak `accept()`.
3. Funkcja `listen()` wskazuje gotowość do zaakceptowania wysyłanych przez klienta żądań połączenia. Po wywołaniu funkcji `listen()` alarm jest ustawiany na uruchomienie po pięciu sekundach. Ten alarm lub sygnał informuje użytkownika o zablokowaniu wywołania funkcji `accept()`.
4. Funkcja `accept()` akceptuje żądanie połączenia od klienta. Wywołanie to zwykle blokuje proces na czas nieokreślony, lecz ponieważ ustawiony został alarm, wywołanie zostanie zablokowane jedynie na czas pięciu sekund. Kiedy alarm się włączy, funkcja `accept` zakończy się z kodem powrotu `-1` i wartością `errno` równą `EINTR`.
5. Funkcja `close()` zamyka wszystkie otwarte deskryptory gniazd.

Uwaga: Korzystając z przykładowego kodu, użytkownik akceptuje warunki określone w temacie “Informacje dotyczące licencji na kod” na stronie 192.

```

/*****
/* 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 przechwytyjąca sygnał. Będzie ona wywoływana, kiedy */
/* pojawi się sygnał. */
*****/
void catcher(int sig)
{
    printf("Wywołanie procedury przechwytyjącej dla sygnału %d\n", sig);
}

/*****
/* Program główny */
*****/
int main (int argc, char *argv[])
{
    struct sigaction sact;
    struct sockaddr_in  addr;
    time_t t;
    int sd, rc;

/*****
/* Utworzenie gniazda AF_INET, SOCK_STREAM */
*****/
    printf("Tworzenie gniazda TCP\n");
    sd = socket(AF_INET, SOCK_STREAM, 0);
    if (sd == -1)
    {
        perror("utworzenie gniazda nie powiodło się");
        return(-1);
    }

/*****
/* Powiązanie gniazda. Nie podano numeru portu, ponieważ */
/* z gniazdem tym nie będzie nawiązywane połączenie. */
*****/
    memset(&addr, 0, sizeof(addr));
    addr.sin_family = AF_INET;
    printf("Wiązanie gniazda\n");
    rc = bind(sd, (struct sockaddr *)&addr, sizeof(addr));
    if (rc != 0)
    {
        perror("powiązanie nie powiodło się");
        close(sd);
        return(-2);
    }

/*****
/* Wykonanie czynności nasłuchiwanie przez gniazdo. */
*****/
    printf("Ustawienie parametru backlog nasłuchiwanie\n");
    rc = listen(sd, 5);
    if (rc != 0)
    {
        perror("nasłuchiwanie nie powiodło się");
        close(sd);
        return(-3);
    }

/*****
/* Ustawienie alarmu, który włączy się po pięciu sekundach. */
*****/
    printf("\nUstawienie alarmu, który włączy się po 5 sek. Alarm spowoduje,\n");
    printf("że zablokowana funkcja accept() zwróci -1 i wartość errno równą EINTR.\n\n");
    sigemptyset(&sact.sa_mask);
    sact.sa_flags = 0;
    sact.sa_handler = catcher;

```

```

sigaction(SIGALRM, &sact, NULL);
alarm(5);

/*****
/* Wyświetla bieżący czas z chwili ustawienia alarmu */
*****/
time(&t);
printf("Przed funkcją accept() godzina %s", ctime(&t));

/*****
/* Akceptowanie wywołania. Normalnie wywołanie to blokowałoby */
/* przez czas nieograniczony, ale alarm powoduje ograniczenie */
/* blokowania do 5 s. Kiedy alarm się włączy, funkcja accept */
/* zakończy się z -1 i wartością errno równą EINTR. */
*****/
errno = 0;
printf("Oczekiwanie na połączenie przychodzące\n");
rc = accept(sd, NULL, NULL);
printf("Funkcja accept() zakończona. rc = %d, errno = %d\n", rc, errno);
if (rc >= 0)
{
printf("Odebrano połączenie przychodzące\n");
close(rc);
}
else
{
perror("łańcuch errno");
}

/*****
/* Wyświetlenie godziny włączenia alarmu */
*****/
time(&t);
printf("Po funkcji accept(), godzina %s\n", ctime(&t));
close(sd);
return(0);
}

```

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 43

Funkcje API asynchronicznych operacji we/wy udostępniają metodę dla wątków modelu klient serwer, umożliwiającą realizację współbieżnych operacji we/wy z efektywnym wykorzystaniem pamięci.

Odsyłacze pokrewne

“Przykład: korzystanie z asynchronicznych operacji we/wy” na stronie 111

Aplikacja tworzy port we/wy zakończenia za pomocą funkcji API `QsoCreateIOCompletionPort()`.

Informacje pokrewne

`accept()`

`listen()`

`close()`

`socket()`

`bind()`

Przykłady: użycie rozsyłania grupowego z obsługą AF_INET

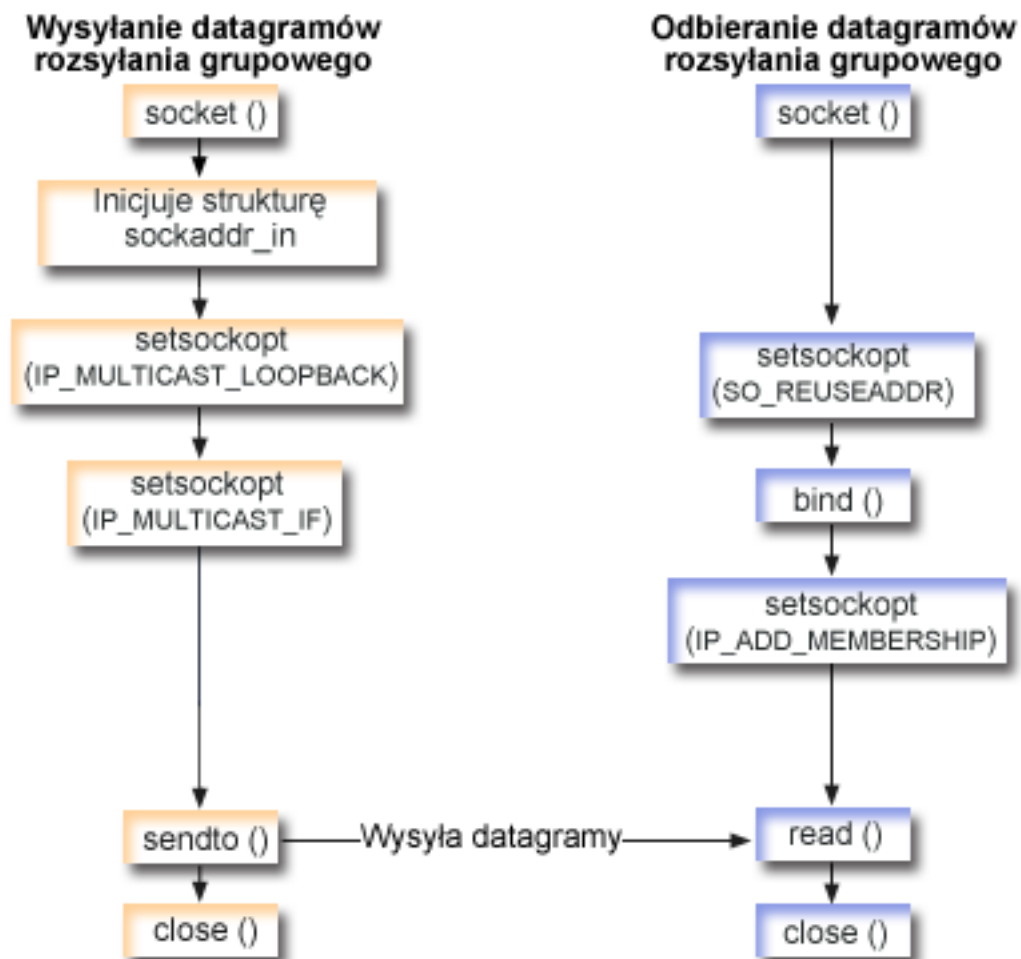
Rozsyłanie grupowe IP zapewnia aplikacjom możliwość wysyłania jednego datagramu IP, który odbierany jest przez grupę hostów w sieci.

Hosty należące do grupy mogą znajdować się w tej samej podsieci lub w różnych podsieciach połączonych routerami obsługującymi rozsyłanie grupowe. Hosty mogą być dołączane do grup i usuwane z nich w każdej chwili. Nie ma ograniczeń dotyczących położenia ani liczby członków grupy hostów. Grupę hostów identyfikuje klasa D adresów internetowych w zakresie od 224.0.0.1 do 239.255.255.255.

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

- IP_ADD_MEMBERSHIP: Umożliwia dołączenie do podanej grupy rozsyłania grupowego.
- IP_DROP_MEMBERSHIP: Umożliwia usunięcie z podanej grupy rozsyłania grupowego.
- IP_MULTICAST_IF: Konfiguruje interfejs, poprzez który wysyłane są wychodzące datagramy rozsyłania grupowego.
- IP_MULTICAST_TTL: Ustawia wartość Time To Live (TTL) w nagłówku IP wychodzącego datagramu rozsyłania grupowego.
- IP_MULTICAST_LOOP: Określa, czy kopia wychodzącego datagramu rozsyłania grupowego ma być dostarczana do hosta wysyłającego tak długo, dopóki jest on członkiem grupy rozsyłania grupowego.

Uwaga: Gniazda systemu i5/OS obsługują rozsyłanie grupowe IP dla rodziny adresów AF_INET.



Przebieg zdarzeń w gnieździe: wysyłanie datagramów rozsyłania grupowego

Na poniższej ilustracji wyjaśniono sekwencję wywołań funkcji gniazd. Opisano także relacje pomiędzy dwiema aplikacjami, które wysyłają i odbierają datagramy rozsyłania grupowego. Kolejny przykład wykorzystuje następującą sekwencję wywołań funkcji:

1. Funkcja `socket()` zwraca deskryptor gniazda odpowiadający punktowi końcowemu. Instrukcja ta informuje również, że dla tego gniazda zostanie użyta rodzina adresów INET (Internet Protocol) z transportem UDP transport (`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 `setsockopt()` ustawia opcję gniazda `IP_MULTICAST_LOOP`, aby system wysyłający nie otrzymywał wysłanych przezeń datagramów rozsyłania grupowego.
4. Funkcja `setsockopt()` używa opcji gniazda `IP_MULTICAST_IF`, która definiuje interfejs lokalny, przez który będą przesyłane datagramy rozsyłania grupowego.
5. Funkcja `sendto()` wysyła datagramy rozsyłania grupowego do podanych grupowych adresów IP.
6. Funkcja `close()` zamyka wszystkie otwarte deskryptory gniazd.

Przebieg zdarzeń w gnieździe: odbieranie datagramów rozsyłania grupowego

Kolejny przykład wykorzystuje następującą sekwencję wywołań funkcji:

1. Funkcja `socket()` zwraca deskryptor gniazda odpowiadający punktowi końcowemu. Instrukcja ta informuje również, że dla tego gniazda zostanie użyta rodzina adresów INET (Internet Protocol) z transportem UDP transport (`SOCK_DGRAM`). Gniazdo to wysyła datagramy do drugiej aplikacji.
2. Funkcja `setsockopt()` ustawia opcję gniazda `SO_REUSEADDR`, umożliwiającą wielu aplikacjom odbieranie datagramów skierowanych do portu lokalnego o tym samym numerze.
3. Funkcja `bind()` określa numer portu lokalnego. W tym przykładzie adres IP jest podany w postaci `INADDR_ANY`, aby możliwe było odbieranie datagramów przeznaczonych dla grupy.
4. Funkcja `setsockopt()` używa opcji gniazda `IP_ADD_MEMBERSHIP`, łączącej grupę, dla której datagramy są przeznaczone. Dołączając do grupy należy podać adres grupy klasy D razem z adresem IP lokalnego interfejsu. System musi wywołać opcję gniazda `IP_ADD_MEMBERSHIP` dla każdego lokalnego interfejsu, który odbiera datagramy rozsyłania grupowego. W tym przykładzie grupa rozsyłania (225.1.1.1) jest połączona z interfejsem lokalnym 9.5.1.1.

Uwaga: Opcję `IP_ADD_MEMBERSHIP` należy wywołać dla każdego interfejsu lokalnego, przez który datagramy rozsyłania grupowego mają być odbierane.

5. Funkcja `read()` odczytuje wysłane datagramy rozsyłania grupowego.
6. Funkcja `close()` zamyka wszystkie otwarte deskryptory gniazd.

Pojęcia pokrewne

“Rozsyłanie grupowe IP” na stronie 60

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

Odsyłacze pokrewne

“Przykład: wysyłanie datagramów rozsyłania grupowego” na stronie 169

Ten przykład umożliwia gniazdu odbieranie datagramów rozsyłania grupowego.

Informacje pokrewne

`close()`

`socket()`

`bind()`

`setsockopt`

`read()`

sendto()

Przykład: wysyłanie datagramów rozsyłania grupowego

Ten przykład umożliwia gniazdu odbieranie datagramów rozsyłania grupowego.

Uwaga: Korzystając z przykładowego kodu, użytkownik akceptuje warunki określone w temacie “Informacje dotyczące licencji na kod” na stronie 192.

```
#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 (Wysyłanie datagramu rozsyłania grupowego) */
    /*                                           */
    /* ----- */

    /*
     * Utworzenie gniazda datagramu, do którego *
     * datagram ma być wysłany.
     */
    sd = socket(AF_INET, SOCK_DGRAM, 0);
    if (sd < 0) {
        perror("otwieranie gniazda datagramowego");
        exit(1);
    }

    /*
     * Inicjowanie struktury grupy sockaddr z
     * adresem grupy 225.1.1.1 i portem 5555.
     */
    memset((char *) &groupSock, 0, sizeof(groupSock));
    groupSock.sin_family = AF_INET;
    groupSock.sin_addr.s_addr = inet_addr("225.1.1.1");
    groupSock.sin_port = htons(5555);

    /*
     * Wyłączenie pętli zwrotnej, aby nie otrzymywać własnych datagramów.
     */
    {
        char loopch=0;

        if (setsockopt(sd, IPPROTO_IP, IP_MULTICAST_LOOP,
            (char *)&loopch, sizeof(loopch)) < 0) {
            perror("ustawienie IP_MULTICAST_LOOP:");
            close(sd);
            exit(1);
        }
    }

    /*
     * Skonfigurowanie lokalnego interfejsu dla wychodzących datagramów
     * rozsyłania grupowego.
     */
}
```

```

* Podany adres IP musi być przypisany do lokalnego interfejsu,
* który obsługuje rozsyłanie grupowe.
*/
localInterface.s_addr = inet_addr("9.5.1.1");
if (setsockopt(sd, IPPROTO_IP, IP_MULTICAST_IF,
              (char *)&localInterface,
              sizeof(localInterface)) < 0) {
    perror("konfigurowanie lokalnego interfejsu");
    exit(1);
}

/*
* Wysłanie komunikatu do grupy rozsyłania grupowego podanej przez
* strukturę groupSock sockaddr.
*/
datalen = 10;
if (sendto(sd, databuf, datalen, 0,
           (struct sockaddr*)&groupSock,
           sizeof(groupSock)) < 0)
{
    perror("wysłanie komunikatu datagramu");
}
}

```

Odsyłacze pokrewne

“Przykłady: użycie rozsyłania grupowego z obsługą AF_INET” na stronie 166

Rozsyłanie grupowe IP zapewnia aplikacjom możliwość wysyłania jednego datagramu IP, który odbierany jest przez grupę hostów w sieci.

Przykład: odbieranie datagramów rozsyłania grupowego

Ten przykład umożliwia gniazdu odbieranie datagramów rozsyłania grupowego.

Uwaga: Korzystając z przykładowego kodu, użytkownik akceptuje warunki określone w temacie “Informacje dotyczące licencji na kod” na stronie 192.

```

#include <sys/types.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <netinet/in.h>
#include <stdio.h>
#include <stdlib.h>

struct sockaddr_in    localSock;
struct ip_mreq        group;
int                  sd;
int                  datalen;
char                  databuf[1024];

int main (int argc, char *argv[])
{
    /* -----*/
    /*                                           */
    /* Przykład kodu RMD (Odebranie datagramu rozsyłania grupowego)*/
    /*                                           */
    /* -----*/

    /*
    * Utworzenie gniazda, z którego *
    * datagram ma być odebrany.
    */
    sd = socket(AF_INET, SOCK_DGRAM, 0);
    if (sd < 0) {

```



```

    perror("otwieranie gniazda datagramowego");
    exit(1);
}

/*
 * Włącz SO_REUSEADDR, aby umożliwić wielu instancjom tej aplikacji
 * odbieranie kopii datagramów rozsyłania grupowego.
 */
{
    int reuse=1;

    if (setsockopt(sd, SOL_SOCKET, SO_REUSEADDR,
                  (char *)&reuse, sizeof(reuse)) < 0) {
        perror("ustawianie SO_REUSEADDR");
        close(sd);
        exit(1);
    }
}

/*
 * Powiąż odpowiedni numer portu z adresem IP
 * podanym jako INADDR_ANY.
 */
memset((char *) &localSock, 0, sizeof(localSock));
localSock.sin_family = AF_INET;
localSock.sin_port = htons(5555);
localSock.sin_addr.s_addr = INADDR_ANY;

if (bind(sd, (struct sockaddr*)&localSock, sizeof(localSock))) {
    perror("wiązanie gniazda datagramu");
    close(sd);
    exit(1);
}

/*
 * Dołącz do grupy rozsyłania grupowego 225.1.1.1 w lokalnym
 * interfejsie
 * 9.5.1.1. Zauważ, że opcja IP_ADD_MEMBERSHIP musi być wywołana
 * dla każdego interfejsu lokalnego, przez który datagramy
 * rozsyłania grupowego mają być odbierane.
 */
group.imr_multiaddr.s_addr = inet_addr("225.1.1.1");
group.imr_interface.s_addr = inet_addr("9.5.1.1");
if (setsockopt(sd, IPPROTO_IP, IP_ADD_MEMBERSHIP,
              (char *)&group, sizeof(group)) < 0) {
    perror("dodawanie grupy rozsyłania grupowego");
    close(sd);
    exit(1);
}

/*
 * Odczyt z gniazda.
 */
datalen = sizeof(databuf);
if (read(sd, databuf, datalen) < 0) {
    perror("odczyt datagramu z komunikatem");
    close(sd);
    exit(1);
}
}

```

Przykład: odpytywanie i aktualizacja serwera DNS

Poniższy przykład ilustruje, jak wysyłać zapytania do systemu nazw domen (DNS) i jak aktualizować jego rekordy.

Uwaga: Korzystając z przykładowego kodu, użytkownik akceptuje warunki określone w temacie “Informacje dotyczące licencji na kod” na stronie 192.

```
/* **** */
/* Ten program aktualizuje DNS przy użyciu sygnatury transakcji (TSIG), */
/* służącej do podpisywania pakietów aktualizacyjnych. Następnie odpytuje */
/* serwer DNS, aby sprawdzić powodzenie aktualizacji. */
/* **** */

/* **** */
/* Wymagane przez program pliki nagłówkowe. */
/* **** */
#include <stdio.h>
#include <errno.h>
#include <arpa/inet.h>
#include <resolv.h>
#include <netdb.h>

/* **** */
/* Deklaracja rekordów aktualizacji - rekordu strefy, rekordu wstępnego */
/* i 2 rekordów aktualizacji. */
/* **** */
ns_updrec update_records[] =
{
    {
        {NULL,&update_records[1]},
        {NULL,&update_records[1]},
        ns_s_zn, /* rekord strefy */
        "mydomain.ibm.com.",
        ns_c_in,
        ns_t_soa,
        0,
        NULL,
        0,
        0,
        NULL,
        NULL,
        0
    },
    {
        {&update_records[0],&update_records[2]},
        {&update_records[0],&update_records[2]},
        ns_s_pr, /* rekord wstępny */
        "mypc.mydomain.ibm.com.",
        ns_c_in,
        ns_t_a,
        0,
        NULL,
        0,
        ns_r_nxdomain, /* rekord nie może istnieć */
        NULL,
        NULL,
        0
    },
    {
        {&update_records[1],&update_records[3]},
        {&update_records[1],&update_records[3]},
        ns_s_ud, /* rekord aktualizujący */
        "mypc.mydomain.ibm.com.",
        ns_c_in,
        ns_t_a, /* adres IPv4... */
        10,
        (unsigned char *)"10.10.10.10",
        11,
        ns_uop_add, /* ...który ma zostać dodany */
        NULL,
        NULL,
    }
}
```

```

        0
    },
    {
        {&update_records[2],NULL},
        {&update_records[2],NULL},
        ns_s_ud,          /* rekord aktualizujący */
        "mypc.mydomain.ibm.com.",
        ns_c_in,
        ns_t_aaaa,        /* adres IPv4... */
        10,
        (unsigned char *)"fedc:ba98:7654:3210:fedc:ba98:7654:3210",
        39,
        ns_uop_add,       /* ...który ma zostać dodany */
        NULL,
        NULL,
        0
    }
};

/*****
/* Dwie poniższe struktury definiują klucz i klucz tajny, które muszą */
/* być zgodne ze skonfigurowanymi w serwerze DNS: */
/* allow-update { */
/* key my-long-key.; */
/* } */
/* */
/* Musi to być binarny równoważnik tajnego klucza base64 */
/* */
*****/
unsigned char secret[18] =
{
    0x6E,0x86,0xDC,0x7A,0xB9,0xE8,0x86,0x8B,0xAA,
    0x96,0x89,0xE1,0x91,0xEC,0xB3,0xD7,0x6D,0xF8
};

ns_tsig_key my_key = {
    "my-long-key",          /* Ten klucz musi istnieć na serwerze DNS */
    NS_TSIG_ALG_HMAC_MD5,
    secret,
    sizeof(secret)
};

void main()
{
    /*****
    /* Definicje zmiennych i struktur. */
    *****/
    struct state res;
    int result, update_size;
    unsigned char update_buffer[2048];
    unsigned char answer_buffer[2048];
    int buffer_length = sizeof(update_buffer);

    /* Wyłączenie opcji init w celu zainicjowania struktury */
    res.options &= ~ (RES_INIT | RES_XINIT);

    result = res_ninit(&res);

    /* Umieszczenie przetwarzania w tym miejscu pozwoli sprawdzić wyniki */
    /* i obsłużyć błędy */

    /* Budowanie buforu aktualizacji (pakietu do wysłania) na podstawie */
    /* rekordów aktualizacji */
    update_size = res_nmkupdate(&res, update_records,
                               update_buffer, buffer_length);

    /* Umieszczenie przetwarzania w tym miejscu pozwoli sprawdzić wyniki */

```

```

/* i obsłużyć błędy */
{
    char zone_name[NS_MAXDNAME];
    size_t zone_name_size = sizeof zone_name;
    struct sockaddr_in s_address;
    struct in_addr addresses[1];
    int number_addresses = 1;

/* Znalezienie autorytatywnego serwera DNS dla domeny, która ma być */
/* aktualizowana */
    result = res_findzonecut(&res, "mypc.mydomain.ibm.com", ns_c_in, 0,
        zone_name, zone_name_size,
        addresses, number_addresses);

/* Umieszczenie przetwarzania w tym miejscu pozwoli sprawdzić wyniki */
/* i obsłużyć błędy */
/* Sprawdzenie, czy znaleziony serwer DNS jest jednym z używanych */
    s_address.sin_addr = addresses[0];
    s_address.sin_family = res.nsaddr_list[0].sin_family;
    s_address.sin_port = res.nsaddr_list[0].sin_port;
    memset(s_address.sin_zero, 0x00, 8);

    result = res_nisourserver(&res, &s_address);

/* Umieszczenie przetwarzania w tym miejscu pozwoli sprawdzić wyniki */
/* i obsłużyć błędy */
/* Ustawienie adresu DNS znalezionej jako res_findzonecut */
/* w strukturze res. Do tego serwera DNS zostanie wysłana */
/* aktualizacja z podpisem TSIG. */
res.nscount = 1;
    res.nsaddr_list[0] = s_address;

/* Wysłanie aktualizacji DNS z podpisem TSIG */
    result = res_nsendsigned(&res, update_buffer, update_size,
        &my_key,
        answer_buffer, sizeof answer_buffer);

/* Umieszczenie przetwarzania w tym miejscu pozwoli sprawdzić wyniki */
/* i obsłużyć błędy */
}

/*****
/* res_findzonecut(), res_nmkupdate(), i res_nsendsigned() można */
/* zastąpić jednym wywołaniem res_nupdate() z użyciem */
/* update_records[1] w celu pominięcia rekordu strefy: */
/* */
/* result = res_nupdate(&res, &update_records[1], &my_key); */
/* */
*****/
/*****
/* Sprawdzenie, czy aktualizacja rzeczywiście powiodła się! */
/* Wybrano protokół TCP, a nie UDP, więc po zainicjowaniu zmiennej res */
/* należy ustawić odpowiednią opcję. Ponadto program będzie ignorował */
/* lokalną pamięć podręczną i zawsze wysyłał zapytania do serwera DNS. */
*****/

res.options |= RES_USEVC|RES_NOCACHE;

/* Wysłanie zapytania o rekordy adresu mypc.mydomain.ibm.com */
result = res_nquerydomain(&res, "mypc", "mydomain.ibm.com.",
ns_c_in, ns_t_a,
    update_buffer, buffer_length);

```

```

/* Przykład obsługi błędów i drukowania komunikatów o błędach          */
if (result == -1)
{
    printf("\nZapytanie nie powiodło się. Wynik = %d \nerrno: %d: %s \
\nh_errno: %d: %s",
        result,
        errno, strerror(errno),
        h_errno, hstrerror(h_errno));
}
/*****
/* Komunikat o błędzie będzie wyglądał następująco:          */
/*
/* zapytanie o domenę nie powiodło się. Wynik = -1          */
/* errno: 0: Brak błędu.                                     */
/* h_errno: 5: Nieznany host                                 */
*****/
return;
}

```

Pojęcia pokrewne

“Ochrona wątków” na stronie 57

Funkcja realizuje ochronę wątków, jeśli można ją uruchomić jednocześnie w wielu wątkach w ramach tego samego procesu.

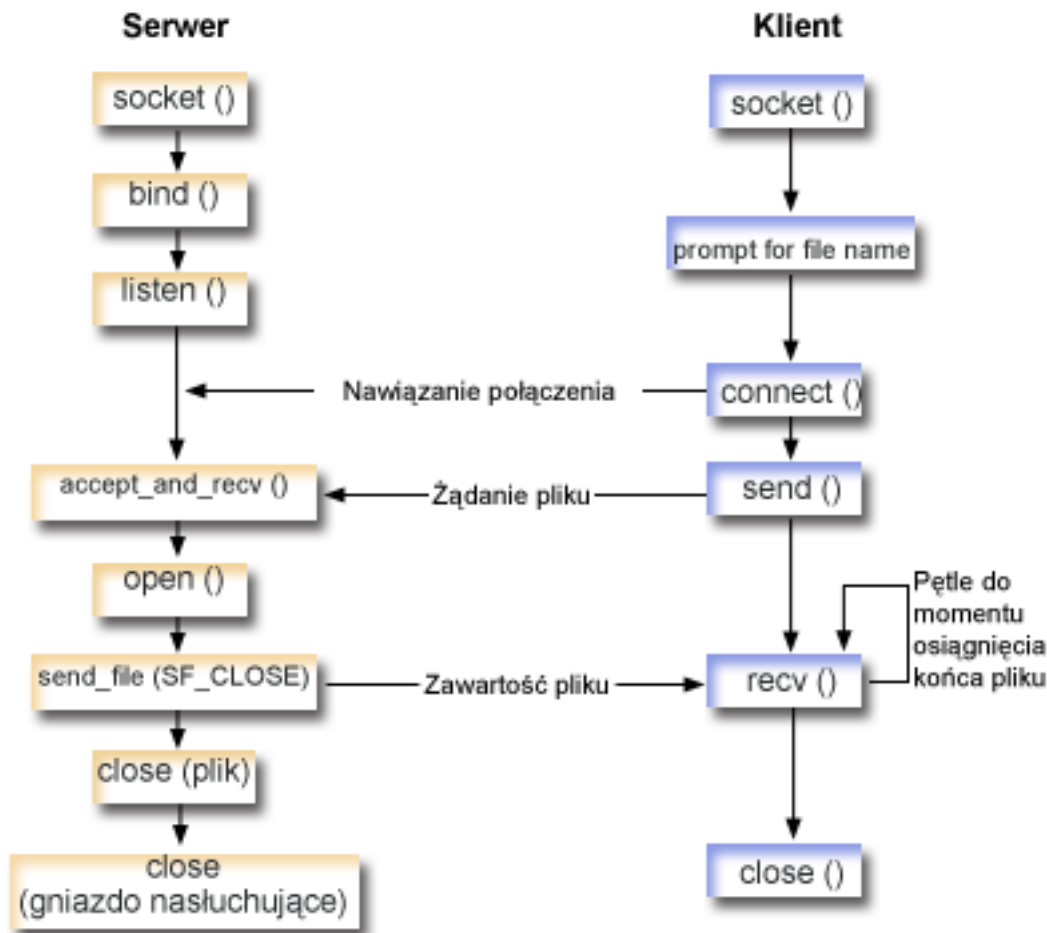
Odsyłacze pokrewne

“Buforowanie danych” na stronie 66

Buforowanie danych odpowiedzi na zapytania do serwera DNS jest wykonywane przez gniazda systemu i5/OS w celu zmniejszenia ruchu w sieci. Pamięć podręczna jest dodawana i aktualizowana, gdy jest to wymagane.

Przykład: przesyłanie danych za pomocą funkcji `send_file()` i `accept_and_recv()`

Poniższe przykłady umożliwiają serwerowi komunikację z klientem za pomocą funkcji `send_file()` i `accept_and_recv()`.



Przebieg zdarzeń w gnieździe: serwer wysyła zawartość pliku

Na poniższej ilustracji wyjaśniono sekwencję wywołań funkcji gniazd. Opisano także relacje pomiędzy dwiema aplikacjami, które wysyłają i odbierają pliki. Kolejny przykład wykorzystuje następującą sekwencję wywołań funkcji:

1. Serwer tworzy gniazdo nasłuchujące, wywołując funkcje `socket()`, `bind()` i `listen()`.
2. Serwer inicjuje struktury adresu lokalnego i zdalnego.
3. Serwer wywołuje funkcję `accept_and_recv()` umożliwiającą oczekiwanie na połączenie przychodzące i pierwszy bufor danych przekazany poprzez to połączenie. Wywołanie to zwraca liczbę odebranych bajtów oraz adres lokalny i zdalny przypisane do tego połączenia. Jest ono kombinacją funkcji `accept()`, `getsockname()` i `recv()`.
4. Serwer otwiera plik, którego nazwa została uzyskana od aplikacji klienta jako dane z funkcją `accept_and_recv()` poprzez wywołanie funkcji `open()`.
5. Funkcja `memset()` zeruje wszystkie pola struktury `sf_parms`. Serwer wstawia w polu deskryptora pliku wartość zwróconą przez funkcję `open()`. Następnie serwer podaje wartość pola wielkości pliku wynoszącą `-1`, co wskazuje, że serwer powinien przesłać cały plik. System wysyła cały plik, więc nie trzeba podawać wartości pola pozycji w pliku.

6. Serwer przesyła zawartość pliku za pomocą funkcji `send_file()`. Funkcja `send_file()` nie kończy działania, dopóki cały plik nie zostanie wysłany lub funkcja nie zostanie przerwana. Funkcja `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()`. Flaga `SF_CLOSE` informuje funkcję `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 podana została opcja `SF_CLOSE`.

Przebieg zdarzeń w gnieździe: klient żąda pliku

Kolejny przykład wykorzystuje następującą sekwencję wywołań funkcji:

1. Klient pobiera nie więcej niż dwa parametry.
Pierwszy parametr (jeśli podany) jest adresem IP w postaci dziesiętnej z kropkami lub nazwą hosta, w którym znajduje się aplikacja serwera.
Drugi parametr (jeśli podany) jest nazwą pliku, który klient próbuje pobrać z serwera. Aplikacja serwera wysyła do klienta zawartość podanego pliku. Jeśli użytkownik nie poda żadnego parametru, klient jako adresu IP serwera użyje parametru `INADDR_ANY`. Jeśli użytkownik nie poda drugiego parametru, program zażąda wpisania nazwy pliku.
2. Klient wywołuje funkcję `socket()` w celu uzyskania deskryptora gniazda.
3. Wywołanie przez klienta funkcji `connect()` w celu nawiązania połączenia z serwerem. Adres IP serwera uzyskano w pierwszym kroku.
4. Wywołanie przez klienta funkcji `send()` w celu podania serwerowi nazwy pliku, który ma być przesłany. Nazwę tę uzyskano w pierwszym kroku.
5. Klient przechodzi do pętli "do" wywołującej funkcję `recv()` do czasu, aż przetwarzanie osiągnie koniec pliku. Kod powrotu 0 w funkcji `recv()` oznacza, że serwer zamknął połączenie.
6. Wywołanie przez klienta funkcji `close()` w celu zamknięcia gniazda.

Pojęcia pokrewne

"Przesyłanie danych pliku - `send_file()` i `accept_and_recv()`" na stronie 61

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

Informacje pokrewne

`accept()`
`recv()`
`send()`
`listen()`
`close()`
`socket()`
`bind()`
`getsockname()`
`open()`
`read()`
`connect()`

Przykład: użycie funkcji API `accept_and_recv()` i `send_file()` do wysyłania zawartości pliku

Poniższe przykłady umożliwiają serwerowi komunikację z klientem za pomocą funkcji `send_file()` i `accept_and_recv()`.

Uwaga: Korzystając z przykładowego kodu, użytkownik akceptuje warunki określone w temacie "Informacje dotyczące licencji na kod" na stronie 192.

```

/*****/
/* 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;

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

    /*****/
    /* Ustaw bit SO_REUSEADDR, aby nie trzeba było */
    /* czekać 2 minuty zanim serwer zostanie ponownie*/
    /* uruchomiony */
    /*****/
    rc = setsockopt(listen_sd,
                    SOL_SOCKET,
                    SO_REUSEADDR,
                    (char *)&flag,
                    sizeof(flag));

    if (rc < 0)
    {
        perror("Niepowodzenie accept()");
        close(listen_sd);
        exit(-1);
    }

    /*****/

```



```

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

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

/*****/
/* Zainicjowanie długości adresu lokalnego */
/* i zdalnego */
/*****/
local_addr_length = sizeof(local_addr);
remote_addr_length = sizeof(remote_addr);

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

/*****/
/* Przejdź przez pętlę raz dla każdego połączenia*/
/*****/
for (i=0; i < num; i++)
{
    /*****/
    /* Oczekiwanie na połączenie przychodzące */
    /*****/
    printf("Iteracja: %d\n", i+1);
    printf(" oczekiwanie na accept_and_recv()\n");

    rc = accept_and_recv(listen_sd,
                        &accept_sd,
                        (struct sockaddr *)&remote_addr,
                        &remote_addr_length,
                        (struct sockaddr *)&local_addr,
                        &local_addr_length,
                        &buffer,
                        sizeof(buffer));

    if (rc < 0)
    {
        perror("Niepowodzenie accept_and_recv()");
        close(listen_sd);
        close(accept_sd);
        exit(-1);
    }
    printf(" Żądanie pliku: %s\n", buffer);
}

```

```

/*****/
/* Otwórz plik do pobrania */
/*****/
fd = open(buffer, O_RDONLY);
if (fd < 0)
{
    perror("Niepowodzenie open()");
    close(listen_sd);
    close(accept_sd);
    exit(-1);
}

/*****/
/* Zainicjowanie struktury sf_parms */
/*****/
memset(&parms, 0, sizeof(parms));
parms.file_descriptor = fd;
parms.file_bytes = -1;

/*****/
/* Zainicjowanie licznika całkowitej liczby */
/* przesłanych bajtów */
/*****/
total_sent = 0;

/*****/
/* Pętla, aż do przesłania całego pliku */
/*****/
do
{
    rc = send_file(&accept_sd, &parms, SF_CLOSE);
    if (rc < 0)
    {
        perror("Niepowodzenie send_file()");
        close(fd);
        close(listen_sd);
        close(accept_sd);
        exit(-1);
    }
    total_sent += parms.bytes_sent;
} while (rc == 1);

printf("Całkowita liczba przesłanych bajtów: %d\n", total_sent);

/*****/
/* Zamknij wysłany plik */
/*****/
close(fd);
}

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

/*****/
/* Zamknij gniazdo akceptujące */
/*****/
if (accept_sd != -1)
    close(accept_sd);
}

```

Informacje pokrewne

send_file()

accept_and_recv()

Przykład: klient żądający pliku

Poniższy przykład umożliwia programowi klienta zażądanie pliku z serwera i zaczekanie, aż serwer w odpowiedzi odeśle zawartość pliku.

Uwaga: Korzystając z przykładowego kodu, użytkownik akceptuje warunki określone w temacie “Informacje dotyczące licencji na kod” na stronie 192.

```
/* ***** */
/* Przykładowy klient żądający pliku od serwera */
/* ***** */
#include <ctype.h>
#include <stdio.h>
#include <stdlib.h>
#include <netdb.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

#define SERVER_PORT 12345

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

    char   filename[256];
    char   buffer[32 * 1024];

    struct sockaddr_in  addr;
    struct hostent      *host_ent;

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

    /* ***** */
    /* Określ nazwę hosta i adres IP maszyny, na     */
    /* której działa serwer                          */
    /* ***** */
    if (argc < 2)
    {
        addr.sin_addr.s_addr = htonl(INADDR_ANY);
    }
    else if (isdigit(*argv[1]))
    {
        addr.sin_addr.s_addr = inet_addr(argv[1]);
    }
    else
    {
        host_ent = gethostbyname(argv[1]);
        if (host_ent == NULL)
        {
            printf("Host nie znaleziony!\n");
            exit(-1);
        }
        memcpy((char *)&addr.sin_addr.s_addr,
               host_ent->h_addr_list[0],
               host_ent->h_length);
    }

    /* ***** */
    /* Sprawdź, czy użytkownik podał w wierszu komend */
    /* nazwę pliku                                     */
    /* ***** */
}
```

```

if (argc == 3)
{
    strcpy(filename, argv[2]);
}
else
{
    printf("Wpisz nazwę pliku:\n");
    gets(filename);
}

/*****/
/* Utwórz gniazdo strumienia AF_INET */
/*****/
sockfd = socket(AF_INET, SOCK_STREAM, 0);
if (sockfd < 0)
{
    perror("Niepowodzenie socket()");
    exit(-1);
}
printf("Gniazdo zakończyło działanie.\n");

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

/*****/
/* Wysłanie żądania do serwera */
/*****/
rc = send(sockfd, filename, strlen(filename) + 1, 0);
if (rc < 0)
{
    perror("Niepowodzenie send()");
    close(sockfd);
    exit(-1);
}
printf("Wysłano żądanie %s\n", filename);

/*****/
/* Odebranie pliku z serwera */
/*****/
do
{
    rc = recv(sockfd, buffer, sizeof(buffer), 0);
    if (rc < 0)
    {
        perror("Niepowodzenie recv()");
        close(sockfd);
        exit(-1);
    }
    else if (rc == 0)
    {
        printf("Koniec pliku\n");
        break;
    }
    printf("Otrzymano bajtów: %d\n", rc);
} while (rc > 0);

```

```

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

```

Narzędzie Xsockets

Narzędzie Xsockets jest jednym z wielu narzędzi dostarczanych z systemem iSeries. Wszystkie narzędzia znajdują się w bibliotece QUSRTOOL. Narzędzie Xsockets umożliwia programistom interaktywną pracę z funkcjami API gniazd.

Narzędzie Xsockets umożliwia realizację następujących zadań:

- zapoznanie się z funkcjami API gniazd,
- interaktywne odtworzenie konkretnych scenariuszy, co ułatwia debugowanie.

Uwaga: Narzędzie Xsockets jest dostarczane jako gotowy produkt, w formacie niezmiennym.

Wymagania wstępne dla narzędzia Xsockets

Przed rozpoczęciem pracy z narzędziem Xsockets należy wykonać następujące czynności:

- instalacja języka ILE C/400.
- instalacja elementu System Openness Includes (opcja 13) z licencjonowanego pakietu IBM i5/OS
- instalacja licencjonowanego programu Serwera IBM HTTP dla i5/OS (5722-DG1)

- Uwaga:** Program ten jest potrzebny do korzystania z narzędzia Xsockets za pośrednictwem przeglądarki WWW.
- Licencjonowany program IBM Developer Kit for Java (5722-JV1) jest zainstalowany.

Uwaga: Program ten jest potrzebny do korzystania z narzędzia Xsockets za pośrednictwem przeglądarki WWW.

Konfigurowanie Xsockets

Istnieją dwie wersje tego narzędzia. Pierwsza wersja jest zintegrowana z klientem iSeries. Wersja ta jest w całości tworzona za pomocą pierwszego zestawu instrukcji. Druga wersja jako klienta używa przeglądarki WWW.

Aby móc korzystać z klienta opartego na przeglądarce WWW, należy najpierw wykonać czynności związane z konfigurowaniem wersji zintegrowanej.

Aby utworzyć narzędzie Xsockets, wykonaj następujące czynności:

1. Aby rozpakować narzędzie, wpisz:

```
CALL QUSRTOOL/UNPACKAGE ('*ALL ' 1)
```

w wierszu komend.

Uwaga: Pomiędzy rozpoczynającym a kończącym znakiem apostrofu (') musi być 10 znaków.

2. Aby do listy bibliotek dodać bibliotekę QUSRTOOL, wpisz

```
ADDLIB QUSRTOOL
```

w wierszu komend.

3. Utwórz bibliotekę, w której zostaną utworzone pliki programu Xsocket, wpisując

```
CRTLIB <nazwa-biblioteki>
```

w wierszu komend. Parametr <nazwa-biblioteki> określa bibliotekę, w której mają być utworzone obiekty narzędzia Xsockets. Na przykład:

CRTLIB MYXSOCKET

jest poprawną nazwą.

Uwaga: Obiektów narzędzia Xsocket nie należy dodawać do biblioteki QUSRTOOL. Może to powodować konflikt z innymi narzędziami w tym katalogu.

4. Aby do listy bibliotek dodać tę bibliotekę, w wierszu komend wpisz

```
ADDLIBLE <nazwa-biblioteki>
```

Parametr <nazwa-biblioteki> określa bibliotekę utworzoną w Kroku 3. Na przykład, aby określić bibliotekę MYXSOCKET, wpisz:

```
ADDLIBLE MYXSOCKET
```

.

5. Utwórz program instalacyjny TSOCRT, który automatycznie zainstaluje narzędzie Xsockets, wpisując:

```
CRCLPGM <nazwa-biblioteki>/TSOCRT QUSRTOOL/QATTCL
```

Parametr <nazwa-biblioteki> określa bibliotekę utworzoną w Kroku 3. Na przykład, aby określić bibliotekę MYXSOCKET, wpisz:

```
CRCLPGM MYXSOCKET/TSOCRT QUSRTOOL/QATTCL
```

.

6. Aby wywołać program instalacyjny, w wierszu komend wpisz:

```
CALL TSOCRT nazwa-biblioteki
```

W miejscu parametru nazwa-biblioteki użyj biblioteki utworzonej w Kroku 3. Na przykład, aby określić bibliotekę MYXSOCKET, wpisz:

```
CALL TSOCRT MYXSOCKET
```

Uwaga: Może to potrwać kilka minut.

Jeśli użytkownik uruchamiający program TSOCRT nie ma uprawnień specjalnego do zarządzania zadaniem (*JOBCTL), funkcja gniazda givedescriptor() zwróci błędy podczas próby przekazania deskryptora do innego zadania, niż właśnie uruchomione.

Program TSOCRT tworzy program CL, program ILE C/400 (dwa moduły), dwa programy serwisowe ILE C/400 (dwa moduły), i trzy zbiory ekranowe. Należy dodać bibliotekę do listy bibliotek przez każdorazowym uruchomieniem narzędzia Xsocket. Wszystkie obiekty utworzone przez narzędzie będą miały nazwę z przedrostkiem TSO.

Szczegółowe informacje dotyczące narzędzia Xsockets znajdują się w sekcji Korzystanie ze zintegrowanego narzędzia Xsocket.

Uwaga: Wersja zintegrowana nie obsługuje funkcji API GSKit gniazd chronionych. Aby napisać program gniazd korzystający z tych funkcji, należy użyć wersji opartej na przeglądarce WWW.

Pojęcia pokrewne

“Używanie Xsockets” na stronie 189

Obecnie można pracować z narzędziem Xsockets na dwa sposoby. Można użyć rodzimego klienta lub przeglądarki WWW.

Zadania pokrewne

“Korzystanie ze zintegrowanego narzędzia Xsockets” na stronie 189

Niniejszy temat zawiera przydatne informacje dotyczące obsługi narzędzia Xsocket na zintegrowanym kliencie.

Co tworzy zintegrowana konfiguracja Xsocket

Tabela zawiera listę obiektów tworzonych przez program instalacyjny. Wszystkie obiekty rezydują w podanej bibliotece.

Tabela 19. 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 pomiędzy JSP a TSOSTSOC.
TSODLT	TSODLT	QATTCL	*PGM	CLP	Program CL do usuwania obiektów narzędzia i/lub podzbiorów zbioru źródłowego.
TSOXSOCK	nie dotyczy	nie dotyczy	*PGM	C	Główny program używany przez interaktywne narzędzie SOCKETS.
TSOXGJOB	nie dotyczy	nie dotyczy	*SRVPGM	C	Program serwisowy służy do obsługi interaktywnego narzędzia SOCKETS.
TSOJNI	nie dotyczy	nie dotyczy	*SRVPGM	C	Program serwisowy stanowiący interfejs pomiędzy JSP a TSOSTSOC, służący do obsługi interaktywnego narzędzia SOCKETS.
TSOXSOCK	TSOXSOCK	QATTSYSC	*MODULE	C	Moduł używany podczas tworzenia programu TSOXSOCK. Plik ten zawiera procedurę main().
TSOSTSOC	TSOSTSOC	QATTSYSC	*MODULE	C	Moduł używany podczas tworzenia programu TSOXSOCK. Zbiór źródłowy zawiera procedury wywołujące funkcje gniazd.
TSOXGJOB	TSOXGJOB	QATTSYSC	*MODULE	C	Moduł używany podczas tworzenia programu TSOXGJOB. Zbiór źródłowy zawiera procedurę identyfikującą zadanie wewnętrzne. Ten wewnętrzny identyfikator zadania składa się z nazwy zadania, ID użytkownika i numeru zadania.

Tabela 19. Obiekty utworzone podczas instalacji narzędzia Xsocket (kontynuacja)

Nazwa obiektu	Nazwa podzbioru	Nazwa zbioru źródłowego	Typ obiektu	Rozszerzenie	Opis
TSODSP	TDSPDSP	QATDDDS	*FILE	DSPF	Zbiór ekranowy używany przez narzędzie Xsockets do tworzenia głównego ekranu zawierającego funkcje gniazd.
TSEFUN	TSEFUN	QATDDDS	*FILE	DSPF	Zbiór ekranowy używany przez narzędzie Xsockets do obsługi różnych funkcji gniazd.
TSOMNU	TDSOMNU	QATDDDS	*FILE	DSPF	Zbiór ekranowy używany przez narzędzie Xsockets do obsługi paska menu.
QATTIFS2	nie dotyczy	nie dotyczy	*FILE	PF-DTA	Zawiera plik JAR używany przez serwer Tomcat.

Konfigurowanie Xsockets do korzystania z przeglądarki WWW

Następujący zestaw instrukcji umożliwia dostęp do narzędzia Xsockets z przeglądarki WWW. Aby utworzyć wiele instancji serwera, można zastosować te instrukcje w tym samym systemie wielokrotnie. Dzięki temu będzie jednocześnie uruchomionych wiele wersji, nasłuchujących na różnych portach.

Pojęcia pokrewne

“Używanie Xsockets” na stronie 189

Obecnie można pracować z narzędziem Xsockets na dwa sposoby. Można użyć rodzimego klienta lub przeglądarki WWW.

Konfigurowanie serwera HTTP (opartego na Apache)

Poniżej zostało opisane konfigurowanie serwera HTTP (opartego na Apache), umożliwiające używanie narzędzia Xsockets za pośrednictwem przeglądarki WWW.

Aby skonfigurować przeglądarkę WWW do współpracy z narzędziem Xsockets, należy najpierw przeprowadzić konfigurację narzędzia Xsockets.

1. Sprawdź, czy instancja admin serwera HTTP pracuje w podsystemie QHTTPSVR. Jeśli nie, można ją uruchomić komendą CL
STRTPSVR SERVER(*HTTP) HTTPSVR(*ADMIN)
2. W przeglądarce WWW wpisz:
http://<nazwa systemu>:2001/.

gdzie<nazwa systemu>jest nazwą systemu iSeries. Na przykład: http://myiSeries:2001/.
3. Na stronie Zadania i5/OS wybierz **Zadządzanie siecią IBM i5/OS**.
4. Z górnego menu wybierz zakładkę **Konfiguracja**.
5. Kliknij **Utwórz nowy serwer HTTP**.
6. Wpisz nazwą instancji serwera. Na przykład, For example, jeśli ta instancja obsługuje narzędzie Xsockets w przeglądarce, można użyć nazwy xsocket. Kliknij **Dalej**.
7. Kliknij **Dalej**, aby zaakceptować domyślny katalog główny serwera.
8. Kliknij **Dalej**, aby zaakceptować domyślny katalog główny dokumentu.

- Wybierz adres IP i dostępny port, którego chcesz użyć. Użyj portu o numerze większym od 1024. Kliknij **Dalej**.

Uwaga: Nie należy wybierać domyślnego portu o numerze 80.

- Wybierz **tak** lub **nie**, aby określić, czy chcesz dla tego serwera utworzyć protokół dostępu. Kliknij .
- Wybierz domyślny czas przechowywania protokołów. Kliknij .
- Na następnej stronie zostanie wyświetlona konfiguracja serwera HTTP (opartego na Apache). Jeśli te ustawienia są poprawne, kliknij **Zakończ**.

Konfigurowanie serwera Tomcat

Po skonfigurowaniu instancji serwera HTTP (opartego na Apache) należy skonfigurować serwer Tomcat, aby uruchomić narzędzie Xsockets w przeglądarce WWW.

- W nagłówku **Manage**, wybierz stworzony przez siebie serwer HTTP.
- W nagłówku **Dynamic content** w lewym panelu wybierz opcję **ASF Tomcat Setup Server Task** .
- Wybierz **Enable servlets for this HTTP Server**. Spowoduje to wpisanie danych do pliku definicji procesów roboczych. Kliknij .
- Na stronie **Workers Definition** zaakceptuj wartości domyślne i kliknij .
- Na stronie **URL to Worker Mapping** kliknij **Add**.
- W kolumnie **URL(Mount Point)** wpisz /xsock. Kliknij **Continue**.
- Kliknij przycisk **Dodaj**.
- W kolumnie **URL(Mount Point)** wpisz /xsock/*. Kliknij **Continue**.
- Kliknij .
- Na stronie **In-Process Application Context Definition** kliknij **Add**.
- W kolumnie **URL Path** wpisz /xsock.
- W kolumnie **Application Base Directory** wpisz webapps/xsock.
- Kliknij **Continue**. Wyświetlony zostanie komunikat informujący o tym, że nie trzeba podawać dalszych informacji.
- Na stronie **Configure Application** kliknij **Configure**.
- W nowo otwartym oknie przeglądarki w polu **Session Object timeout** wybierz 3 dni.

Uwaga: Jest to wartość zalecana; parametr **Session Object timeout** może jednak przyjmować dowolne wartości.

- Kliknij **Add**, aby dodać definicję serwletu i wykonać następujące czynności:
 - W polu **Servlet class name** wpisz com.ibm.iseries.xsocket.XSocketServlet.
 - W polu **URL patterns** wpisz /*.
 - Ustaw wartość **Startup load sequence** na 3.
 - Kliknij **Continue**.
 - Kliknij **OK**. Spowoduje to zamknięcie okna przeglądarki.
- W głównym oknie konfigurowania serwera Tomcat kliknij .
- Kliknij **Finish**.
- Kliknij **OK**. Konfigurowanie serwera Tomcat do obsługi narzędzia Xsockets dobiegło końca.

Aktualizacja plików konfiguracyjnych

Po skonfigurowaniu serwera Tomcat na serwerze HTTP (opartym na Apache) do obsługi narzędzia Xsockets należy ręcznie dokonać zmian w kilku plikach konfiguracyjnych dla instancji. Należy zaktualizować trzy zbiory: web.xml, JAR i httpd.conf.

Do wykonania tych czynności będą potrzebne następujące informacje:

- Nazwa biblioteki zawierającej zbiory aplikacji Xsockets. Zostały one utworzone podczas wstępnego konfigurowania narzędzia Xsockets jako klienta rodzimego.
- Nazwa serwera utworzonego podczas konfigurowania serwera HTTP (opartego na Apache).

1. Zaktualizuj zbiór web.xml
 - a. W wierszu komend wpisz
wrklnk '/www/<server_name>/webapps/xsock/WEB-INF/web.xml'

gdzie <nazwa serwera> jest nazwą instancji serwera utworzonej podczas konfigurowania serwera Apache. Na przykład, jeśli nazwą serwera jest xsocks, wpisz:

```
wrklnk '/www/xsocks/webapps/xsock/WEB-INF/web.xml'
```

- b. Aby przeprowadzić edycję pliku, naciśnij 2.
- c. Znajdź wiersz </servlet-class> w zbiorze web.xml.
- d. Pod tym wierszem wpisz następujący kod:

```
<init-param>
    <param-name>library</param-name>
    <param-value>XXXX</param-value>
</init-param>
```

W miejscu znaków XXXX wpisz nazwę biblioteki utworzonej podczas konfigurowania narzędzia Xsockets.

- e. Zapisz plik i zakończ sesję edycji.
2. Przenieś plik JAR W wierszu komend wpisz komendę:
CPY OBJ('/QSYS.LIB/XXXX.LIB/QATTIFS2.FILE/TSOXSOCK.MBR')
TOOBJ('/www/<nazwa serwera>/webapps/xsock/WEB-INF/lib/tsoxsock.jar')
FROMCCSID(*OBJ) TOCCSID(819) OWNER(*NEW)

gdzie XXXX jest nazwą biblioteki utworzoną podczas konfigurowania narzędzia Xsockets, a <nazwa_serwera> jest nazwą instancji serwera utworzoną podczas konfigurowania serwera HTTP (opartego na Apache).

3. **Opcjonalne:** Dodaj sprawdzanie uprawnień do zbioru httpd.conf. Wymusza to uwierzytelnianie użytkowników próbujących skorzystać z aplikacji Xsockets przez serwer Apache.

Uwaga: Niezbędne są także uprawnienia do zapisu podczas tworzenia gniazd UNIX.

- a. W wierszu komend wpisz
wrklnk '/www/<nazwa serwera>/conf/httpd.conf'

gdzie <nazwa serwera> jest nazwą instancji serwera utworzonej podczas konfigurowania serwera Apache. Na przykład, jeśli nazwą serwera jest xsocks, wpisz:

```
wrklnk '/www/xsocks/conf/httpd.conf'
```

- b. Aby przeprowadzić edycję pliku, naciśnij 2.
- c. Poniższe wiersze wstaw na końcu pliku.

```
<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.

Testowanie narzędzia Xsockets w przeglądarce WWW

Po zakończeniu ręcznej aktualizacji plików konfiguracyjnych można przetestować narzędzie Xsockets w przeglądarce.

1. Aby uruchomić instancję serwera, w wierszu komend wpisz:
STRTCPSVR SERVER(*HTTP) HTTPSVR(<nazwa serwera>)

gdzie <nazwa serwera> jest nazwą instancji serwera utworzonej podczas konfiguracji serwera Apache.

Uwaga: Może to zająć trochę czasu.

2. Sprawdź status, uruchamiając komendę WRKACTJOB z interfejsu wiersza komend. Jeśli wszystkie zadania z server_name mają status SIGW, można przejść do następnego kroku.
3. W przeglądarce WWW wpisz następujący adres URL:
`http://<nazwa systemu>:<port>/xsock/index`

gdzie <nazwa systemu> i <port> to nazwa instancji serwera i numer portu określone podczas konfiguracji serwera Apache.

4. Po wyświetleniu zachęty wpisz nazwę użytkownika i hasło do serwera. Powinien zostać wyświetlony klient WWW narzędzia Xsocket.

Używanie Xsockets

Obecnie można pracować z narzędziem Xsockets na dwa sposoby. Można użyć rodzimego klienta lub przeglądarki WWW.

Aby pracować ze zintegrowaną wersją narzędzia Xsocket, należy je skonfigurować. Poza konfiguracją narzędzia Xsockets dla zintegrowanego klienta, należy zakończyć kolejne etapy konfiguracji Xsocket aby móc korzystać z przeglądarki internetowej, jeśli użytkownik preferuje pracę z narzędziem w środowisku przeglądarki. Wiele pojęć jest podobnych w obu wersjach narzędzia. Oba narzędzia umożliwiają interaktywne wywoływanie funkcji gniazd i dostarczają informacji o numerach błędów zwracanych przez te funkcje. Jednakże ich interfejsy w pewnym stopniu różnią się.

Uwaga: Aby pracować z programami gniazd korzystającymi z funkcji API GSKit gniazd chronionych, konieczne jest używanie wersji WWW tego narzędzia.

Pojęcia pokrewne

“Konfigurowanie Xsockets do korzystania z przeglądarki WWW” na stronie 186

Następujący zestaw instrukcji umożliwia dostęp do narzędzia Xsockets z przeglądarki WWW. Aby utworzyć wiele instancji serwera, można zastosować te instrukcje w tym samym systemie wielokrotnie. Dzięki temu będzie jednocześnie uruchomionych wiele wersji, nasłuchujących na różnych portach.

Zadania pokrewne

“Konfigurowanie Xsockets” na stronie 183

Istnieją dwie wersje tego narzędzia. Pierwsza wersja jest zintegrowana z klientem iSeries. Wersja ta jest w całości tworzona za pomocą pierwszego zestawu instrukcji. Druga wersja jako klienta używa przeglądarki WWW.

Korzystanie ze zintegrowanego narzędzia Xsockets

Niniejszy temat zawiera przydatne informacje dotyczące obsługi narzędzia Xsocket na zintegrowanym kliencie.

1. W wierszu komend dodaj do listy bibliotek bibliotekę, w której znajduje się narzędzie Xsocket, wpisując komendę
`ADDLIBLE <nazwa-biblioteki>`

gdzie <nazwa-biblioteki> jest nazwą biblioteki utworzonej podczas konfigurowania zintegrowanego klienta Xsockets. Na przykład, jeśli nazwą biblioteki jest MYXSOCKET, wpisz:
`ADDLIBLE MYXSOCKET`
2. W wierszu komend wpisz
`CALL TSOXSOCK`
3. Zostanie wyświetlone okno Xsockets, które umożliwi dostęp do wszystkich procedur gniazd poprzez pasek menu i pola wyboru. Okno to jest zawsze wyświetlane po wybraniu funkcji gniazd. Interfejsu tego można używać do wybierania istniejących programów używających gniazd. Aby utworzyć nowe gniazdo, należy wykonać opisane poniżej czynności:
 - a. Z listy funkcji gniazda wybierz **socket** i naciśnij klawisz Enter.
 - b. W wyświetlonym oknie **socket()** **prompt** wybierz odpowiednią rodzinę adresów, typ gniazda i protokół, a następnie naciśnij klawisz Enter.
 - c. Wybierz **Descriptor**, a następnie **Select descriptor**.

Uwaga: Jeśli istnieją już inne deskryptory gniazd, spowoduje to wyświetlenie listy deskryptorów aktywnych gniazd.

d. Z wyświetlonej listy wybierz utworzony przez siebie deskryptor gniazda.

Uwaga: Jeśli istnieją inne deskryptory gniazd, narzędzie automatycznie stosuje tę funkcję do najnowszego deskryptora gniazda.

4. Z listy funkcji gniazda wybierz tę, z którą chcesz pracować. Zostanie ona uruchomiona dla deskryptora gniazda wybranego w kroku 3c. Po wybraniu funkcji gniazd zostanie wyświetlona grupa okien, w których można wprowadzić informacje właściwe dla danej funkcji gniazd. Jeśli na przykład zostanie wybrana funkcja connect(), w wyświetlonych oknach trzeba będzie podać długość adresu, rodzinę adresów i dane adresu. Wybrana funkcja gniazd jest następnie wywoływana z podanymi informacjami. Wszelkie błędy, jakie wystąpią przy wywołaniu funkcji zostaną wyświetlone jako kody błędów errno.

Uwagi:

1. Narzędzie Xsockets korzysta z graficznego interfejsu dla DDS. Dlatego sposób wprowadzania danych i realizacji wyborów w oknach lub na ekranach zależy od tego, czy używasz terminalu graficznego, czy tekstowego. Na przykład, na terminalu graficznym będzie widoczne pole wyboru funkcji gniazda; w przeciwnym razie będzie to zwykłe pole.
2. Należy wiedzieć, że w gnieździe są dostępne żądania ioctl(), które nie zostały zaimplementowane w narzędziu.

Używanie Xsockets w przeglądarce WWW

Niniejszy temat zawiera pomoc przydatną w korzystaniu z narzędzia Xsockets z poziomu przeglądarki WWW.

Aby pracować z narzędziem Xsockets w przeglądarce WWW, należy przeprowadzić konfigurowanie rodzimej wersji Xsockets i wszystkie czynności związane z konfigurowaniem przeglądarki WWW. Należy się również upewnić, czy włączone są informacje cookie.

1. W przeglądarce WWW wpisz:

```
http://nazwa-serwera:2001/
```

gdzie nazwa-serwera to nazwa systemu iSeries, na którym pracuje instancja serwera.

2. Wybierz **Administration**.
3. Z lewego paska nawigacyjnego wybierz opcję **Manage HTTP Servers**.
4. Wybierz nazwę instancji i kliknij **Start**. Możesz także uruchomić instancję serwera z wiersza komend, wpisując:
`STRTCPSVR SERVER(*HTTP) HTTPSVR(<nazwa instancji>)`

gdzie <nazwa_instancji> jest nazwą serwera HTTP utworzonego podczas konfigurowania serwera Apache. Na przykład można użyć instancji serwera o nazwie xsocks.

5. Aby wyświetlić aplikację Xsockets, w przeglądarce wpisz poniższy adres URL:

```
http://<nazwa systemu>:<port>/xsock/index
```

gdzie <nazwa systemu> to nazwa systemu iSeries a <port> to port podany podczas tworzenia instancji HTTP. Na przykład, jeśli nazwą systemu jest myiSeries, a instancja serwera HTTP nasłuchuje na porcie 1025, możesz wpisać:

```
http://myiSeries:1025/xsock/index
```

6. Po załadowaniu narzędzia Xsocket w przeglądarce można pracować z istniejącym deskryptorem gniazda lub utworzyć nowe. Wiele pojęć jest podobnych w obu wersjach narzędzia. Oba narzędzia umożliwiają interaktywne wywoływanie funkcji gniazd i dostarczają informacji o numerach błędów zwracanych przez te funkcje. Jednakże ich interfejsy w pewnym stopniu różnią się. Aby utworzyć nowy deskryptor gniazd, należy wykonać opisane poniżej czynności:
 - a. W menu **Xsocket** wybierz **gniazdo**.
 - b. W wyświetlonym oknie **Xsocket Query** wybierz odpowiednią rodzinę adresów, typ gniazda i protokół dla tego deskryptora gniazda. Kliknij **Submit**.

- c. Po ponownym załadowaniu strony, nowy deskryptor gniazda zostanie wyświetlony w menu rozwijanym **Socket**.
- d. W menu **Xsocket** wybierz wywołania funkcji, które mają być stosowane dla tego deskryptora gniazda. Podobnie jak w zintegrowanej wersji Xsockets, jeśli nie wybierany zostanie deskryptor gniazda, narzędzie zastosuje wywołania funkcji do najnowszego deskryptora gniazda.

Usuwanie obiektów utworzonych przez narzędzie Xsocket

Może zająć potrzeba usunięcia obiektów utworzonych przez narzędzie Xsockets. Program instalacyjny tworzy program o nazwie TSODLT, który służy do usuwania obiektów utworzonych przez Xsockets (z wyjątkiem biblioteki i samego programu TSODLT) lub usuwania podzbiorów źródłowych utworzonych przez Xsockets albo przez obra programy.

Opisany poniżej zestaw komend umożliwia usuwanie obiektów:

Aby usunąć TYLKO podzbiory źródłowe używane przez narzędzie, wpisz:

```
CALL TSODLT (*YES *NONE)
```

Aby usunąć TYLKO obiekty utworzone przez narzędzie, wpisz:

```
CALL TSODLT (*NO nazwa-biblioteki)
```

Aby usunąć ZARÓWNO podzbiory źródłowe, jaki i obiekty utworzone przez narzędzie, wpisz:

```
CALL TSODLT (*YES nazwa-biblioteki)
```

Dostosowywanie Xsockets

Narzędzie Xsockets można zmienić dodając obsługę dla sieciowych procedur gniazd, na przykład dla inet_addr().

W przypadku dostosowywania narzędzia do własnych potrzeb, nie zaleca się dokonywania zmian w bibliotece QUSRTOOL. Należy skopiować zbiory źródłowe do osobnej biblioteki i tam dokonać zmian. Dzięki temu w bibliotece QUSRTOOL zostaną zachowane oryginalne zbiory, na wypadek, gdyby były potrzebne w przyszłości. Do ponownej kompilacji narzędzia po dokonaniu zmian można użyć programu TSOCRT (jeśli zbiory źródłowe zostały skopiowane do osobnej biblioteki, należy również dokonać zmian w programie TSOCRT). Do usunięcia starych wersji obiektów narzędzia przed utworzeniem nowej wersji służy program TSODLT.

Narzędzia do serwisowania

W związku z ciągłym wzrostem zastosowań gniazd i gniazd chronionych w aplikacjach i serwerach używanych na potrzeby e-biznesu, rosną również potrzeby związane z rozwojem narzędzi serwisowych.

Udoskonalone narzędzia serwisowe umożliwiają śledzenie działania programów używających gniazd w celu znalezienia rozwiązań problemów związanych z aplikacjami używającymi gniazd i warstwy SSL. Narzędzia te umożliwiają programistom i pracownikom centrum obsługi umiejscowienie problemów z użyciem gniazd poprzez wybór takich parametrów gniazd, jak adres IP lub informacje o porcie.

Poniższa tabela stanowi przegląd informacji o tych narzędziach serwisowych.

Tabela 20. 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 do tylko tych gniazd, które mają ustawioną opcję SO_DEBUG. Od wersji V5R2 śledzenie Licencjonowanego Kodu Wewnętrznego można filtrować pod kątem wątku, zadania, profilu użytkownika, nazwy zadania i nazwy serwera.

Tabela 20. Narzędzia serwisowe dla gniazd i gniazd chronionych (kontynuacja)

Narzędzie serwisowe	Opis
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 innych, niezwiązanych z gniazdami punktów śledzenia. Dane te zawierają kod zakończenia i kod błędu w przypadku wystąpienia błędu podczas operacji gniazda.
Śledzenie rejestratora przebiegu przetwarzania	Informacje ze śledzenia komponentu Licencjonowanego Kodu Wewnętrznego gniazd będą teraz obejmowały zrzut pozycji rejestratora przebiegu przetwarzania dla każdej wykonanej operacji gniazda.
Powiązane informacje o zadaniu	Umożliwiają pracownikom serwisowym i programistom znalezienie wszystkich zadań powiązanych z gniazdem połączonym lub nasłuchującym. Dla aplikacji używających gniazd z rodziny adresów AF_INET lub AF_INET6 informacje te mogą być wyświetlone za pomocą komendy NETSTAT.
Status połączenia (Opcja 3 komendy NETSTAT) udostępnia parametr SO_DEBUG	Udostępnia rozszerzone informacje debugowania niskiego poziomu, jeśli dla aplikacji używającej gniazd zostanie ustawiona opcja SO_DEBUG.
Kod zakończenia i przetwarzanie komunikatów gniazd chronionych	Wyświetla standardowe komunikaty kodów powrotu gniazd chronionych przy użyciu dwóch funkcji API SSL_. Funkcje te to SSL_Sterror() i SSL_Perror(). Ponadto funkcja gsk_sterror() udostępnia podobne możliwości co funkcje API GSKit. Dostępna jest także funkcja API hsterror(), dostarczająca informacji o kodach powrotu procedur programu tłumaczącego.
Śledzenie danych dotyczących wydajności	Udostępnia informacje ze śledzenia przepływu danych z aplikacji poprzez gniazda i stos TCP/IP.

Informacje pokrewne

SSL_Sterror()

SSL_Perror()

gsk_sterror()

hsterror()

Komenda STRTRC (Rozpoczęcie śledzenia)

Informacje dotyczące licencji na kod

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.

I Z UWZGLĘDNIENIEM BEZWZGLĘDNIEM OBOWIĄZUJĄCYCH GWARANCJI, KTÓRYCH NIE MOŻNA
I WYKLUCZYĆ, IBM, PROGRAMIŚCI I DOSTAWCY IBM NIE UDZIELAJĄ, W ZAKRESIE PROGRAMU CZY
I WSPARCIA TECHNICZNEGO, JAKICHKOLWIEK GWARANCJI (W TYM TAKŻE RĘKOJMI) ANI NIE
I USTALAJĄ WARUNKÓW, ZARÓWNO WYRAŻNYCH JAK I DOMNIEMANYCH, A W SZCZEGÓLNOŚCI
I GWARANCJI CZY WARUNKÓW PRZYDATNOŚCI HANDLOWEJ, PRZYDATNOŚCI DO OKREŚLONEGO
I CELU CZY NIENARUSZANIA PRAW STRON TRZECICH.

I W ŻADNYM PRZYPADKU IBM, PROGRAMIŚCI CZY DOSTAWCY IBM NIE PONOSZĄ
I ODPOWIEDZIALNOŚCI ZA NASTĘPUJĄCE STRATY LUB SZKODY, NAWET JEŚLI BYLIBY
I POINFORMOWANI O MOŻLIWOŚCI ICH WYSTĄPIENIA:

- I 1. UTRATA LUB USZKODZENIE DANYCH;
- I 2. SZKODY BEZPOŚREDNIE, SZCZEGÓLNE, UBOCZNE, POŚREDNIE ORAZ SZKODY, KTÓRYCH NIE
I MOŻNA BYŁO PRZEWIDZIEĆ PRZY ZAWIERANIU UMOWY; ORAZ

| 3. UTRATA ZYSKÓW, KONTAKTÓW HANDLOWYCH, PRZYCHODÓW, REPUTACJI (GOODWILL) CZY
| 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 przysyłać 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 podmiotów 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.

Licencjodawcy 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 niniejszej publikacji oraz wszystkie inne licencjonowane materiały dostępne dla
- | tego programu są dostarczane przez IBM na warunkach określonych w Umowie IBM z Klientem, Międzynarodowej
- | Umowie Licencyjnej IBM na Program lub w innych podobnych umowach zawartych między IBM i użytkownikami.

Wszelkie dane dotyczące wydajności zostały zebrane w kontrolowanym środowisku. W związku z tym rezultaty uzyskane w innych środowiskach operacyjnych mogą się znacząco różnić. Niektóre pomiary mogły być dokonywane na systemach będących w fazie rozwoju i nie ma gwarancji, że pomiary te wykonane na ogólnie dostępnych systemach dadzą takie same wyniki. Niektóre z pomiarów mogły być estymowane przez ekstrapolację. Rzeczywiste wyniki mogą być inne. Użytkownicy powinni we własnym zakresie sprawdzić odpowiednie dane dla ich środowiska.

Informacje dotyczące produktów innych podmiotów uzyskano od dostawców tych produktów, z opublikowanych 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 innych podmiotów 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 jakiegokolwiek 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. Kody te nie zostały kompleksowo przetestowane we wszelkich możliwych warunkach. IBM nie może zatem gwarantować ani sugerować niezawodności, użyteczności i funkcjonalności tych programów.

Każda kopia programu przykładowego lub jakiegokolwiek jego fragment, jak też jakiegokolwiek 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
- | C/400
- | i5/OS
- | e(logo)server
- | eServer
- | IBM
- | IBM (logo)
- | iSeries
- | Integrated Language Environment
- | OS/400
- | Redbooks

Windows NT jest znakiem towarowym Microsoft Corporation w Stanach Zjednoczonych i/lub w innych krajach.

Java oraz wszystkie znaki towarowe dotyczące języka Java są znakami towarowymi Sun Microsystems, Inc. w Stanach Zjednoczonych i/lub w innych krajach.

UNIX jest zastrzeżonym znakiem towarowym 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, WYRAŻNYCH CZY DOMNIEMANYCH, A W SZCZEGÓLNOŚCI DOMNIEMANYCH GWARANCJI PRZYDATNOŚCI HANDLOWEJ, PRZYDATNOŚCI DO OKREŚLONEGO CELU ORAZ NIENARUSZANIA PRAW STRON TRZECICH.

IBM