UNIWERSYTET GDAŃSKI
WYDZIAŁ MATEMATYKI I FIZYKI

Gniazda BSD

Jacek Nowicki

Praca magisterska
napisana pod kierunkiem
prof. dra hab. Andrzeja Mostowskiego

Gdańsk 2003

Spis treści

Wstęp

Z każdym dniem Internet staje się coraz popularniejszy, a to pociąga za sobą fakt, że programy sieciowe nabierają większego znaczenia. Kilka lat temu programowanie aplikacji sieciowych było prawdziwym wyzwaniem. Programista musiał znać wiele szczegółów dotyczących sieci, aby napisać prostą aplikację działającą w Internecie. Dzisiaj sytuacja wygląda zupełnie inaczej. Istnieją wygodne narzędzia do programowania sieciowego, których możliwości zależą od systemu operacyjnego i języka programowania. Jednym z nich jest interfejs gniazdowy, zwany też gniazdami BSD (ang. *Berkeley Software Distribution*), wywodzący się z systemu operacyjnego BSD utworzonego na Uniwersytecie w Berkeley. Interfejs gniazd pozwala programiście skoncentrować się na tworzeniu aplikacji, ukrywając trudności związane z programowaniem sieciowym na niskim poziomie.

Celem niniejszej pracy jest przedstawienie teoretycznych i praktycznych podstaw zagadnień dotyczących programowania sieciowego, wykorzystującego interfejs gniazd BSD. Praca składa się z pięciu rozdziałów i jest podzielona na dwie części :

- Pierwsze dwa rozdziały prezentują teoretyczny aspekt tematu. Opisałem w nich historię powstania gniazd BSD (rozdział 1) oraz podstawowe definicje z nimi związane (rozdział 2).

- Kolejne trzy rozdziały składają się na część praktyczną pracy. Przedstawiłem problematykę adresowania gniazd (rozdział 3) i funkcje systemowe wchodzące w skład interfejsu gniazd (rozdział 4). Zaawansowane zagadnienia związane z gniazdami opisałem w rozdziale 5. W tej części pracy umieściłem kilkanaście przykładowych programów, które będą pomocne w zrozumieniu przedstawionej teorii.

Wszystkie przykładowe programy napisałem w języku ANSI C i przetestowałem na komputerze z zainstalowanym systemem LINUX Red Hat 7.3 (Valhalla). Programy te umieściłem na dyskietce dołączonej do pracy.

Rozdział 1. Krótki rys historyczny

4 października 1957 roku Związek Radziecki umieścił na orbicie okołoziemskiej pierwszego sztucznego satelitę o nazwie “Sputnik”, bijąc tym samym Amerykanów w dziedzinie podboju kosmosu. W roku 1962 wybuchł kryzys na Kubie. Te wydarzenia spowodowały, że Stany Zjednoczone zaczęły zabezpieczać się przed ewentualną wojną atomową. Rząd amerykański nie ograniczając wydatków na rozwój nowoczesnych technologii powołał do życia agencję ARPA (Advanced Research Project Agency – Agencję do Zaawansowanych Projektów Technicznych), której celem było stworzenie zdecentralizowanego systemu komunikacji (tak jakby bez punktu centralnego), który działałby nawet po częściowym zniszczeniu w ataku nuklearnym części podłączonych do niego urządzeń. Owocem pracy agencji było powstanie w 1969 roku sieci ARPAnet - pierwszej rozległej sieci stworzonej na potrzeby ARPA. Łączyła centra badań i uczelnie, umożliwiając im pracę nad technologiami sieciowymi. Po pewnym czasie ARPA została przekształcona w agencję DARPA (*ang. Defense Advanced Research Projects Agency – Agencja do spraw Wojskowych Zaawansowanych Projektów Technicznych)*.

Równocześnie z rozwojem ARPAnetu, od roku 1969 rozpoczął się rozwój systemu UNIX. Na Uniwersytecie w Berkeley (*ang. University of California, Berkeley, w skrócie UCB)* utworzono własną odmianę systemu UNIX znaną jako BSD. Rozwijany przez studentów i naukowców system stawał się coraz bardziej funkcjonalny. W miarę upływu czasu pojawiały się jego kolejne wersje, których możliwości i udogodnienia wzrastały w zaskakującym tempie. W efekcie tych prac powstał system **3BSD** będący następcą wersji 1BSD i 2BSD. Ten niesamowity rozwój zwrócił uwagę agencji DARPA, która postanowiła dofinansowywać dalsze badania związane z tym systemem. W 1977 roku UCB i DARPA nawiązały współpracę, której celem było stworzenie systemu operacyjnego zdolnego pracować na licznych platformach sprzętowych używanych przez agencję. Specjalna grupa – Grupa Badań nad Systemami Operacyjnymi (ang. CSRG - *Computer System Research Group*) powołana przez UCB miała na celu dostosowanie systemu BSD do potrzeb agencji. W 1980 roku światło dzienne ujrzał system **4BSD**, a następnie **4.1BSD** ulepszony pod kątem wydajności. Jednak ARPA na tym nie poprzestała i przedłużyła kontrakt zlecając UCB włączenie do systemu obsługę sieci ARPAnet i poprawienie komunikacji międzyprocesowej. Dalekowzroczni programiści “poszli dalej” i włączyli do systemu obsługę innych protokołów sieciowych. Efektem ich pracy było powstanie w 1983 roku systemu **4.2BSD**, wyposażonego w solidne oprogramowanie sieciowe, w którego skład wchodził między innymi **interfejs gniazd BSD**. W roku 1990 wprowadzono kilka zmian do interfejsu gniazdowego w związku z ukazaniem się systemu **4.3BSD Reno**, w którym oprogramowanie protokołów OSI weszło do jądra systemu.

Punktem wyjścia dla wielu wersji systemu UNIX był kod oprogramowania sieciowego jakieś wersji systemu BSD, zawierającego oprogramowanie interfejsu gniazdowego.

Rozdział 2. Wprowadzenie do gniazd

Rozdział ten poświęcony jest kilku kluczowym pojęciom związanym z tematem gniazd BSD. Większość z nich jest doskonale znana programistom pracującym w środowisku UNIX'a.

2.1. Komputer w sieci

System komputerowy ma charakter wspólnoty symbiotycznej – sprzęt i oprogramowanie zależą ściśle od siebie nawzajem i nie mogą działać osobno. Sprzęt składa się z urządzeń peryferyjnych, procesorów, pamięci, dysków i innych urządzeń elektronicznych, które razem stanowią całość komputera. Jednak komputer bez oprogramowania jest bezużyteczny. Oprogramowanie sterujące, niezbędne do działania komputera, nazywamy **systemem operacyjnym**. Jest to niskopoziomowe oprogramowanie obsługujące sprzęt i zapewniające określony zestaw usług programom użytkowym.

Popularnym systemem operacyjnym, który może pracować w sieci jest system **UNIX** lub jego nowoczesna implementacja **LINUX** – wielozoadaniowy i wielodostępowy system operacyjny klasy UNIX, dystrybuowany na podstawie licencji GNU (ang. *GNU General Public License*) opracowanej przez Free Software Foundation. Pomyślany początkowo jako narzędzie dla hobbystów i profesjonalnych programistów, LINUX staje się w coraz większym stopniu systemem operacyjnym dla “przeciętnego użytkownika”. Będąc zarazem systemem wydajnym, szybkim i darmowym, zdobywa domowe i profesjonalne środowiska komputerowe.

Ściśle związany z UNIX'em (LINUX'em) jest **język C**, będący językiem ogólnego stosowania. Charakteryzuje się prostotą wyrażeń, nowoczesnym sterowaniem, nowoczesnymi strukturami danych oraz bogatym zestawem operatorów. Język ten opracował i zrealizował **Dennis Ritchie** dla systemu operacyjnego UNIX działającego na mikrokomputerze DEC PDP-11.

Dysponując komputerem z zainstalowanym systemem operacyjnym (na przykład systemem LINUX), kompilatorem języka C oraz dowolnym edytorem tekstu (na przykład *emacs*) możemy napisać program. **Program** jest to plik wykonywalny, utworzony zazwyczaj przy użyciu programu łączącego i znajdujący się w pamięci dyskowej. Program staje się **procesem** wtedy, kiedy jest wykonywany przez system operacyjny. Często się mówi, że program jest obiektem statycznym, czyli formalnym opisem tego, co ma być wykonane, a proces jest obiektem dynamicznym, czyli ciągiem sekwencyjnie wykonywanych przez system operacyjny instrukcji programu.

W jądrze UNIX'a istnieje pewna liczba (zazwyczaj ograniczona) tak

zwanych **funkcji systemowych** (ang. *shell call*), za pomocą których aktywny proces może uzyskać różne usługi ze strony jądra systemu. Większość funkcji systemowych zwraca wartość całkowitą dodatnią w przypadku pomyślnego wykonania. Jeżeli podczas wykonywania funkcji wystąpi błąd, to funkcja zazwyczaj zwraca wartość -1 , a zmienna globalna ***errno*** otrzyma wartość całkowitą dodatnią, wskazującą na rodzaj błędu. Wszystkim dodatnim wartościom zmiennej *errno* odpowiadają stałe, których nazwy składają się z wielkich liter, zaczynają się od litery E i są zdefiniowane w pliku nagłówkowym <*sys/errno.h*>. Żaden błąd nie ma wartości 0.

2.2. Sieć komputerowa

Sieć komputerowa jest systemem komunikacyjnym łączącym systemy końcowe, zwane stacjami sieciowymi (ang. h*ost computer*) . Komputery są połączone jakimś medium fizycznym, na przykład kablem komunikacyjnych. Niezależnie od sposobu połączenia komputerów ze sobą, celem sieci jest zapewnienie komunikacji między poszczególnymi stacjami – komputerami do niej podłączonymi.

Najpopularniejszą, siecią komputerową jest **Internet** – największa sieć, łącząca sieci komputerowe na całym świecie (“sieć sieci”). Składa się z setek tysięcy kilometrów światłowodów, łącz satelitarnych oraz całej masy mniej lub bardziej skomplikowanym urządzeń. Użytkownicy Internetu, których liczbę szacuje się w milionach, mogą za jego pośrednictwem mieć dostęp do “oceanu” informacji, komunikować się między sobą, wymieniać dane itp.

Większość programów użytkowych, które są przeznaczone do działania w sieci komputerowej, można podzielić na dwie grupy. Pierwsza z nich to aplikacje zwane **serwerami**, które dostarczają usług w Internecie, drugi zbiór składa się z programów zwanych klientami, które z tych korzystają. Termin serwer powoduje czasem nieporozumienia. Formalnie oznacza on program, który czeka biernie na wykonanie pewnej usługi, a nie komputer, który ją wykonuje. Jeśli jednak jakiś komputer jest wyznaczony do wykonywania jednego lub więcej serwerów, to sam jest czasami (niepoprawnie) nazywany serwerem. Producenci sprzętu komputerowego pogłębiają jeszcze te nieporozumienia, gdyż nazywają serwerami komputery o dostatecznie szybkim procesorze, odpowiednio pojemnej pamięci i stosownie wyrafinowanym systemie operacyjnym.

Komunikacja w sieci oparta jest na **protokołach**. Protokoły są to formalne reguły postępowania. Na przykład w stosunkach międzynarodowych, protokoły minimalizują problemy wynikające z różnic kulturowych przy współpracy różnych narodów. Zgadzając się na wspólny zestaw ogólnie przyjętych reguł, które są niezależne od jakichkolwiek zwyczajów narodowych, protokoły dyplomatyczne minimalizują nieporozumienia; każdy wie jak się

zachować i jak interpretować zachowania innych. Podobnie przy łączności komputerów konieczne jest zdefiniowanie reguł, które rządzą komunikacją w sieci. Ponieważ zdefiniowanie takich zasad jest niesłychanie trudne, **Międzynarodowa Organizacja ds. Standardów** (ang. *International Standards Organization [ISO]*) opracowała model architektury służący do opisu zasad komunikacji w sieci. Model **odniesienia łączenia systemów otwartych** (ang. *Open System Interconnect [OSI] Reference Model*) gwarantuje uzyskanie wspólnego mianownika dla zagadnień komunikacyjnych. Wszystko co zostało zdefiniowane za pomocą tego modelu jest ogólnie rozumiane i szeroko stosowane przez osoby zajmujące się komunikacją w sieci. Model OSI składa się z siedmiu warstw na których znajduje się jeden lub więcej protokołów, które opisują sposoby realizacji zadań przeznaczonych dla danej warstwy. Zbiór protokołów obowiązujących na różnych warstwach i mogących stanowić podstawę dla użytecznej sieci nazywamy **rodziną protokołów** (ang. p*rotocol family*).

Rodzinę TCP/IP (ang. *Transmission Control Protocol / Internet Protocol*) będącą rodziną protokołów odpowiedzialną za komunikację w Internecie również możemy przedstawić za pomocą modelu OSI, ale za pomocą czterech warstw : warstwa kanałowa; warstwa sieciowa; warstwa transportowa; warstwa zastosowań. W modelu czterowarstwowym rodzinę protokołów wyznaczają dwie warstwy : transportowa (protokół TCP i protokół UDP) oraz sieciowa (protokół IP). Natomiast w jedną warstwę kanałową są połączone protokoły i cechy charakterystyczne sieci na poziomie jej topologii oraz użytego sprzętu (Ethernet albo Token ring). Przestrzenią programów użytkowych jest zaś warstwa zastosowań.

Do najważniejszych protokołów z rodziny TCP/IP należą :

- Protokół IP (*ang. Internet Protocol*) jest protokołem warstwy sieciowej i jest on fundamentalnym elementem Internetu. Do najważniejszych jego zadań należą między innymi obsługa doręczania pakietów dla protokołów z warstwy transportowej takich jak TCP i UDP oraz definiowanie schematu adresowania w Internecie.

- Protokół TCP (ang. *Transmission Control Protocol*) jest protokołem połączeniowym znajdującym się na warstwie transportowej. Umożliwia niezawodne i w pełni dwukierunkowe (ang. *full-duplex*) przesyłanie strumienia danych. W większości internetowych programów stosuje się ten protokół.

- Protokół UDP (ang. *User Datagram Protocol*) jest protokołem bezpołączeniowym znajdującym się na warstwie transportowej. W odróżnieniu od protokołu TCP, który jest niezawodny, protokół UDP nie daje gwarancji, że dana wiadomość dotrze do wyznaczonego celu.

Aby komputery podłączone do Internetu mogły się ze sobą komunikować, musi istnieć pewien sposób ich identyfikacji. Innymi słowy, każda stacja podłączona do sieci musi mieć przypisany numer identyfikacyjny, który jest niepowtarzalny. Typowy adres stacji składa się z identyfikatora sieci oraz identyfikatora stacji w tej sieci. W Internecie komputery są identyfikowane za pomocą 32-bitowej liczby całkowitej, znanej jako **adres IP**. Adresy te są wyznaczane przez upoważniony do tego węzeł centralny – Sieciowe Centrum Informacyjne czyli NIC (ang. *Network Information Center*), zarządzane przez firmę SRI International. Aby adresy IP były łatwo czytelne, zostały podzielone na 8-bitowe liczby zwane oktetami (format ten często jest nazywany **kropkową notacją czwórkową**). Ponieważ łatwiej zapamiętać nazwy niż liczby, usługa nazewnictwa domen (ang. *Domain Name Service – DNS*) przyporządkowuje unikatowym adresom liczbowym IP nazwy. Na przykład komputer o nazwie *quark.physics.grouch.edu* ma adres IP 0x954C0C04, zapisywany jako *146.76.12.4.* Nie wszystkie adresy sieci i komputerów są dostępne dla użytkowników. Na przykład adresy, których pierwszy bajt jest większy od 223 są zarezerwowane. Nas będzie najbardziej interesował adres **127.0.0.1** będący adresem sieci zwrotnej (ang. *loopback adress)*. Sieć zwrotna składa się z jednego komputera o nazwie *localhost* i adresie *127.0.0.1*. Oznacza to, że samotny komputer (nie podłączony do żadnej sieci) z zainstalowanym systemem UNIX działa w sieci jednoelementowej, którą stanowi ten pojedyczy komputer. W przykładach opisanych w dalszej części pracy użyłem właśnie sieci zwrotnej. Adres takiego komputera można znaleźć w pliku hostów sieciowych */etc/hosts*.

Komputer w sieci może działać jako stacja **jednosieciowa** lub **wielosieciowa**, czyli stacja połączona z co najmniej dwiema sieciami. Każda sieć, z którą porozumiewa się komputer, posiada przypisany jej interfejs sprzętowy, przez który następuje dostęp do sprzętu sieciowego. Komputer może mieć odmienne nazwy w każdej z sieci, a z pewnością będzie miał odrębne adresy. Wypływa z tego wniosek, że każdy adres w Internecie określa w jednoznaczny sposób daną stację, ale nie każda stacja musi mieć dokładnie jeden adres.

Znajomość adresu IP danego komputera, czyli umiejętność zlokalizowania go w sieci nie wystarcza do nawiązania komunikacji, ponieważ na komputerze o danym adresie IP może działać wiele aplikacji pełniących rolę serwera. W celu rozróżnienia tych procesów używa się 16-bitowych numerów portów. Zarówno dla protokołów TCP jak i UDP zdefiniowano grupę portów ogólnie znanych (ang. *well known ports*), przeznaczonych do wykonywania ogólnie znanych usług. Na przykład, w każdej implementacji rodziny protokołów TCP/IP, która pozwala korzystać z protokołu FTP (ang. *File Transfer Protocol*), serwerowi FTP przypisano ogólnie znany port o numerze 21. Protokół TFTP, czyli prymitywny protokół przesyłania plików (ang. *Trivial File Transfer Protocol*), ma w protokole UDP przyporządkowany ogólnie znany port 69. Klienci łącząc się z serwerem używają **portów efemerycznych** (ang. *ephemeral port*), czyli

krótkotrwałych. Numery tych portów są zazwyczaj automatycznie wyznaczane klientom przez protokoły TCP i UDP, które gwarantują, że nie ma portu przypisanego do dwóch procesów jednocześnie i że numer takiego portu jest zawsze większy od numerów dobrze znanych portów. Dokument RFC 1700 zawiera wykaz numerów portów wyznaczonych przez organizację IANA (ang. *Internet Assigned Numbers Authority*). Każdy numer portu należy do jednego z trzech zakresów :

- Porty ogólnie znane (ang. *well-known ports*) mają numery z przedziału 0-1023. Nadzór nad tymi numerami portów i ich przyporządkowanie należy do organizacji IANA. W miarę możności ten sam numer portu jest przypisany do tej samej usługi w obu protokołach – TCP i UDP. Na przykład port 80 przyporządkowano do serwera sieci WWW dla obu protokołów, pomimo że obecnie we wszystkich implementacjach używa się tylko protokołu TCP.

- Porty zarejestrowane (ang. *registered ports*) mają numery z przedziału 1024-49151. Organizacja IANA nie sprawuje nad nimi nadzoru, lecz dla wygody użytkowników sieci rejestruje numery portów i sporządza wykazy ich przyporządkowania. W miarę możności ten sam numer portu jest przypisany do tej samej usługi w obu protokołach – TCP i UDP. Na przykład porty od 6000 do 6063 przyporządkowano do serwera X-Windows dla obu protokołów.

- Porty dynamiczne lub prywatne mają numery z przedziału 49152-65535. Organizacja IANA nic nie mówi o tych portach. Są to porty nazywane portami efemerycznymi.

W systemie LINUX, informacja o tym, jakie usługi odpowiadają jakim portom przechowywana jest w pliku */etc/services*. Do najbardziej znanych usług należą :

- port o numerze 7 - usługa echo.
- port o numerze 9 - usługa discard.
- port o numerze 13 - usługa daytime.
- port o numerze 19 - usługa chargen.
- port o numerze 21 - usługa FTP.
- port o numerze 23 - usługa Telnet.
- port o numerze 25 - serwer pocztowy używający protokołu SMTP.
- port o numerze 37 - usługa time.
- port o numerze 53 - usługa DNS.
- port o numerze 80 - serwer WWW.

2.3 Model komunikacji klient-serwer

Standardowym modelem komunikacji w sieci jest model **"klient-serwer**". Polega on na tym, że jeden proces zwany **serwerem** dostarcza usługi pod określonym adresem IP i numerem portu. Drugi proces zwany **klientem** znając adres IP i numer portu serwera, może skorzystać z jego usługi. Przykładowy scenariusz mógłby wyglądać tak :

- Proces, zwany serwerem, rozpoczyna pracę w pewnym systemie komputerowym. Po zainicjowaniu pracy serwer "zasypia" czekając na proces, zwany klientem, który skontaktuje się z nim, zamawiając jakąś usługę.

- Proces zwany klientem, rozpoczyna pracę albo w tym samym systemie, co serwer, albo w innym systemie połączonym z systemem serwera za pomocą sieci komputerowej. Klient wysyła zamówienie do serwera, prosząc o pewien rodzaj usługi (na przykład serwer musi podać klientowi dokładną godzinę).

- Kiedy serwer zakończy wykonywanie usługi dla klienta, wtedy znowu zasypia i oczekuje na nadejście następnego żądania.

2.4. Definicja gniazda

Kombinacja adresu IP i numeru portu nosi nazwę **gniazda** (ang. *socket*). Można powiedzieć, że gniazdo jest programową abstrakcją używaną do reprezentowania końcówki połączenia między dwoma komputerami. Dla każdego połączenia istnieją gniazda na obydwu uczestniczących w nim komputerach (aplikacjach działających na tych komputerach). Wyobraźmy sobie rozciągający się pomiędzy komputerami kabel, którego końce są wpięte do tych gniazd. Formalnie gniazdo powinniśmy zdefiniować jako zbiór trzyelementowy :

{protokół komunikacyjny, adres-obcy, proces-obcy} - pierwsze gniazdo

{protokół komunikacyjny, adres-lokalny, proces-lokalny} – drugie gniazdo

Wartością *adres-lokalny* i *adres-obcy* są identyfikatory stacji lokalnej oraz stacji odległej. Natomiast elementy *proces-lokalny* i *proces-obcy* określają konkretne procesy działające na obu komputerach. Te trzyelementowe zbiory stanowią punkty końcowe komunikacji w sieci zwane **półasocjacjami**.

Jeżeli rozpocznie się komunikacja między powyższymi komputerami, zostanie utworzony pięcioelementowy zbiór wystarczający do pełnego określenia obu komunikujących się ze sobą procesów.

{protokół, adres-lokalny, proces-lokalny, adres-obcy, proces, obcy}

Ten zbiór nazywamy **asocjacją**.

2.5. Interfejs gniazd

Interfejs wykorzystywany przez program użytkowy przy interakcji z oprogramowaniem protokołów warstwy transportowej nazywa się **interfejsem programu użytkowego** (ang. *Application Program Interface – API*). Interfejs API określa zestaw operacji, które program użytkowy może wykonywać w ramach interakcji z oprogramowaniem protokołów. Większość systemów oprogramowania definiuje interfejs API, podając zestaw funkcji, które program może wywoływać, oraz argumentów, których każda z nich się spodziewa. Zwykle API zawiera oddzielną funkcję dla każdej operacji podstawowej. Interfejs API może na przykład zawierać funkcję, która jest wykorzystywana do ustanawiania komunikacji, oraz inną funkcję, która jest używana do wysyłania danych.

Standardy protokołów komunikacyjnych nie określają zwykle interfejsu, którego programy mają używać przy interakcji z nimi. Protokoły określają ogólne operacje, które powinny być udostępnione, oraz pozwalają systemowi operacyjnemu na zdefiniowanie konkretnego interfejsu API, którego programy będą używać do wykonywania tych operacji. Chociaż standardy protokołów pozwalają projektantom systemów operacyjnych na wybranie interfejsu API, to wielu z nich wybrało **interfejs gniazd BSD**, który stał się standardem interfejsów API.

2.5. Atrybuty gniazd

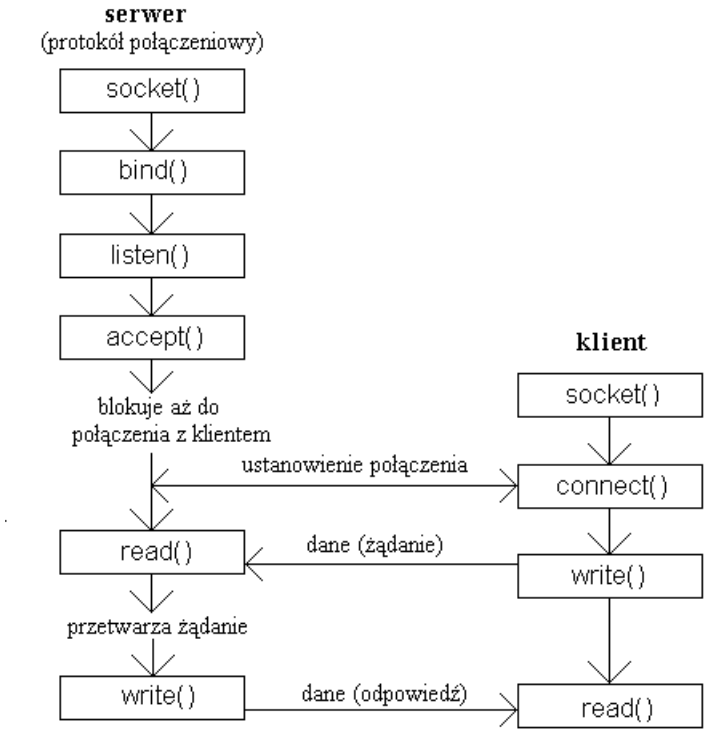
Znając definicje gniazd można zrobić krok naprzód i zająć się podstawowymi parametrami, które mają wpływ na nowo utworzone gniazdo. Gniazda muszą mieć swoją domenę, czyli określone środowisko, w którym będą działać. Powinny mieć typ, czyli sposób komunikowania się w sieci. Zazwyczaj do pełnego zdefiniowania gniazda wystarczają powyższe atrybuty. Czasem zdarzają się sytuację, gdy programista musi ręcznie przypisać gniazdu protokół komunikacyjny. Gniazda mają również swój adres, który jest kojarzony z ich nazwą.

Rodzaj transmisji

Rodzaj transmisji informuje nas w jaki sposób gniazda będą traktować transmitowane dane. Gniazdo może przyjąć jedną z poniższych transmisji :

- **Transmisja połączeniowa**

Transmisja połączeniowa gwarantuje dostarczenie wszystkich pakietów danych w takiej kolejności, w jakiej zostały wysłane. Najpierw jest ustanawiane łącze pomiędzy dwoma komunikującymi się procesami, a dopiero potem zachodzi wymiana danych. Rozwiązanie takie oznacza, że pomiędzy tymi procesami jest wyznaczona trasa oraz że obydwaj uczestnicy transmisji są aktywni. Ustanowienie kanału komunikacyjnego wymaga dodatkowego czasu. Ponadto większość protokołów połączeniowych gwarantuje dostarczenie wiadomości, co jeszcze bardziej wydłuża transmisję, gdyż zachodzi konieczność przeprowadzania obliczeń i weryfikowania poprawności. Protokoły połączeniowe są niezawodne, ale niestety ta niezawodność jest uzyskana kosztem czasu. Transmisja połączeniowa przypomina rozmowę telefoniczną. Najpierw ustanawia się połączenie z rozmówcą, następnie przez pewien czas wymienia się dane i wreszcie zamyka się połączenie.



Rys. 1. Użycie funkcji systemowych w protokole połączeniowym

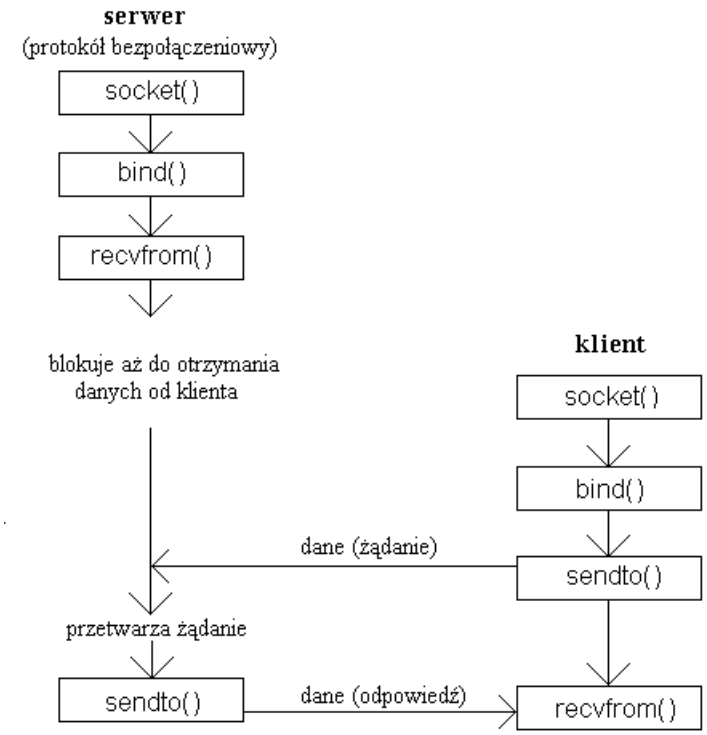
Rozważmy typowy scenariusz transmisji połączeniowej. Serwer jest procesem oczekującym na określoną liczbę połączeń klienta. Jego zadaniem jest obsługa żądań klientów. Serwer musi oczekiwać połączeń pod powszechnie znaną nazwą. Na przykład w protokole TCP/IP nazwą tą będzie adres IP oraz numer portu. Najpierw aplikacja serwera tworzy gniazdo, które jest po prostu zasobem systemu operacyjnego przypisanym do procesu serwera. Służy do tego funkcja systemowa *socket()*. Następnie należy związać to gniazdo z jego powszechnie znaną nazwą (adresem zdefiniowanym w

gniazdowej strukturze adresowej). Zadanie to jest realizowane za pomocą funkcji *bind()*. Teraz należy przełączyć gniazdo w tryb nasłuchiwania, co umożliwia wywołanie systemowe *listen()*. Na zakończenie, gdy klient próbuje nawiązać łączność, serwer może zaakceptować to połączenie za pomocą funkcji *accept()*, która tworzy nowe gniazdo niezależne od gniazda nazwanego i przeznaczone do komunikacji z danym klientem. Od strony klienta sytuacja jest o wiele prostsza, gdyż nawiązanie połączenia wymaga mniejszej liczby kroków. Aplikacja klienta tworzy gniazdo nienazwane używając funkcji *socket()*. Następnie próbuje nawiązać połączenie z serwerem, wykorzystując jego nazwane gniazdo jako adres. Realizuje to używając wywołania systemowego *connect()* . Po nawiązaniu połączenia oba gniazda za pomocą funkcji systemowych *read()* i *write()* zapewniają dwukierunkową komunikację.

- **Transmisja bezpołączeniowa**

Transmisja bezpołączeniowa nie gwarantuje dostarczenia danych ani przekazania ich w odpowiedniej kolejności. Pakiety mogą zostać zgubione lub pomieszane w czasie transmisji w związku z błędami sieci lub z innych powodów. Transmisja bezpołączeniowa przypomina usługi pocztowe. Każda przesyłka zawiera pełny adres odbiorcy. Przesyłki wysyłane pod ten sam adres mogą przebywać różne trasy, zanim zostaną doręczone. Dlatego istnieje możliwość, że dwa listy nadane pod ten sam adres w krótkim odstępie czasu, dotrą do adresata w odwrotnej kolejności. Nie ma też pełnej gwarancji, że przesyłka będzie pomyślnie dostarczona. Programista implementujący gniazda oparte na transmisji bezpołączeniowej musi sam zabezpieczyć swój program przed skutkami ich zawodności. Transmisja bezpołączeniowa jest wielokrotnie szybsza niż połączeniowa. Może być wykorzystywana do takich aplikacji, w których nie ma większego znaczenia to, iż kilka pakietów danych zostanie tu i tam zagubionych, ponieważ szybkość jest dla nich najważniejsza. Przykładem takiej aplikacji może być gra sieciowa przeznaczona dla wielu użytkowników. Każdy gracz stosuje datagramy, aby wysłać do innych graczy informacje o swojej aktualnej pozycji w grze. Jeżeli nawet któryś z pakietów nie dotrze do któregoś z graczy, bardzo szybko otrzyma on następny pakiet, dzięki czemu będzie miał wrażenie płynności gry.

Oto przykład typowego scenariusza transmisji bezpołączeniowej. Aplikacja serwera tworzy gniazdo nienazwane za pomocą funkcji *socket()*. Następnie nadaje mu nazwę używając wywołania systemowego *bind()*. Odmienność polega na tym, że proces serwera nie nasłuchuje ani też nie akceptuje połączeń. Zamiast tego aplikacja serwera po prostu oczekuje na nadchodzące dane. Serwer może odbebrać dane używając funkcji *recvfrom()*. Podobnie klient nie ustanawia połączenia z serwerem, ale po prostu wysyła do niego datagram za pośrednictwem wywołania systemowego *sendto()*.



Rys. 2. Użycie funkcji systemowych w protokole bezpołączeniowym

➢ Domena gniazd

Domena gniazd określa środowisko sieciowe, z którego będą korzystać gniazda oraz sposób zapisu adresów gniazd identyfikujących jeden z końców połączenia komunikacyjnego. Gniazda mogą występować w następujących środowiskach :

- **Gniazda w dziedzinie UNIX'a**

Z gniazd w **dziedzinie UNIX'a** (ang. *Unix domain protocols*) możemy korzystać do komunikowania się z procesami tylko w obrębie tego samego systemu operacyjnego. W tej dziedzinie można implementować zarówno gniazda połączeniowe jak i bezpołączeniowe. Można przyjąć, że obie implementacje są niezawodne, ponieważ znajdują się wewnątrz jądra systemu i nie są przesyłane przez urządzenia zewnętrzne, takie jak linie komunikacyjne łączące komputery. Nie są potrzebne sumy kontrolne ani inne środki zapewniające niezawodność. Protokołem obsługującym gniazda połączeniowe jest *unixstr*, a gniazda bezpołączeniowe – *unixdg*. Adresem gniazda jest nazwa pliku przechowywana w systemie plików.

- **Gniazda w dziedzinie Internetu**

Gniazd z domeny Internetu możemy używać do komunikacji między procesami znajdującymi się na różnych komputerach podłączonych do sieci. W tym środowisku możemy implementować zarówno gniazda połączeniowe jak i

bezpołączeniowe. Używanym protokołem dla gniazd połączeniowych jest TCP (ang. *Transmission Control Protocol*), a dla gniazd bezpołączeniowych – UDP (ang. *User Datagram Protocol*). Oba protokoły działają na bazie protokołu niższego poziomu IP (ang. *Internet Protocol*). Gniazda połączeniowe zapewniają niezawodne dostarczanie danych dzięki pełnodupleksowemu połączeniu implementowanemu za pomocą datagramów IP. Niezawodność uzyskuje się przez użycie mechanizmu *potwierdzeń* i *retransmisji*. Jeżeli nadawca nie dostanie na czas potwierdzenia odebrania pakietu przez odbiorcę, to zakłada, że dane zginęły i retransmituje je. Po pewnej liczbie ponownych retransmisji oprogramowanie TCP zaniecha wysyłania danych, przy czym cały czas przeznaczony na próby wysyłania danych wynosi od 4 do 10 minut (zależnie od implementacji systemu). Gniazda bezpołączeniowe są zawodne, więc jeżeli chcemy uzyskać pewność, że datagram dojdzie do odbiorcy, to musimy nasze oprogramowanie użytkowe uzupełnić o wiele właściwości takich jak potwierdzenie uzyskania danych przez odbiorcę, obsługę czasu oczekiwania na odpowiedź, ponawianie retransmisji itp. Adres gniazda internetowego zajmuje 32 bity i składa się z dwóch części : identyfikatora sieci oraz identyfikatora stacji w tej sieci. Oprócz tego używa się 16-bitowego numeru portu służącego do identyfikacji konkretnego procesu na komputerze o danym adresie IP.

➢ Protokół komunikacyjny

Czasem zdarza się, że mechanizm transportowy potrafi korzystać z różnych protokołów komunikacyjnych, aby określić żądany typ gniazda. Wtedy programista może wskazać najwłaściwszy protokół dla danego gniazda. Zaleca się wyrażenie zgody, aby system operacyjny wybrał domyślny protokół komunikacyjny.

Rozdział 3. Adresowanie gniazd

W tym i następnym rozdziale zajmę się interfejsem gniazd BSD, będącym zestawem funkcji, które mogą być wykorzystane przez program użytkowy do interakcji z oprogramowaniem protokołów warstwy transportowej. Najpierw omówię problematykę adresowania gniazd, co jest chyba najbardziej kłopotliwym aspektem pisania oprogramowania sieciowego.

Gniazda mogą być używane z dowolnymi protokołami, zatem format adresu zależy od tego, który z protokołów będzie użyty. Interfejs gniazd definiuje ogólną postać reprezentacji adresów, a następnie wymaga, aby każda rodzina protokołów określała sposób korzystania z niej przez adresy protokołowe.

3.1. Ogólna gniazdowa struktura adresowa

W 1982 roku została zdefiniowana w pliku nagłówkowym <*sys/socket.h*> ogólna gniazdowa struktura adresowa *sockaddr*.

```
struct sockaddr
{
  sa_family_t sa_family;  // rodzina adresu //
  char sa_data[14] // właściwy adres //
}
```

Obecnie *sa_family_t* to krótka liczba całkowita (ang. *short integer*), której długość w systemie LINUX wynosi dwa bajty. Cała struktura ma 16 bajtów długości. Element *sa_data[14]* struktury reprezentuje 14 bajtów, zawierających dane o adresie.

Ogólna struktura adresu gniazda nie jest specjalnie użyteczna dla programisty. Niemniej dostarcza schematu, do którego muszą się dostosować wszystkie inne struktury adresów.

3.2. Adresowanie gniazd w dziedzinie Internetu

Dla każdej rodziny protokołów zdefiniowano właściwą jej gniazdową strukturę adresową. Nazwy tych struktur zaczynają się od członu *sockaddr_*, po którym występuje drugi człon nazwy, identyfikujący daną rodzinę protokołów. Na przykład rodzina protokołów TCP/IP jako definicji adresu używa struktury *sockaddr_in*, zdefiniowanej w pliku nagłówkowym <*netinet/in.h*>.

```
struct in_addr
{
  unsigned long int s_addr;  // 32 bitowy adres IP
}

struct sockaddr_in
{
  sa_family_t sin_family;  // Tutaj należy wpisać domenę AF_INET //
  unsigned short int sin_port // 16-bitowy numer portu TCP lub UDP //
  struct in_addr sin_addr  // 32-bitowy adres IP //
  char sin_zero[8] // nie używane (wartości zerowe)
}
```

Strukturę powinno się wypełniać w następujący sposób :

- Pole “sin_family” określa rodzinę adresów gniazdowej struktury adresowej. Wypełniając je wartością AF_INET, sprawiamy, że adres jest zgodny z regułami adresowania protokołu IP.

- Pole “sin_port” definiuje numer portu TCP/IP na potrzeby adresu gniazda. Jest to 16-bitowa liczba całkowita bez znaku, która musi być podana w porządku sieciowym (definicję porządku sieciowego omówię w dalszej części rozdziału).

- Pole “sin_addr” przechowuje numer IP hosta, podany w porządku sieciowym. Po dokładnym przyjrzeniu się strukturze *in_addr* można zauważyć, że jest to 32-bitowa liczba całkowita bez znaku. Każdy bajt jest 8-bitową liczbą bez znaku. W ten sposób w każdym bajcie może być zapisana dowolna liczba dziesiętna z przedziału od 0 do 255. Ponieważ jest to wartość bez znaku, liczby nie mogą być ujemne.

- Ostatnie pole struktury definiuje 8-bitowe pole zer. Dzieje się tak, ponieważ TCP/IP wymaga tylko sześć bajtów do zapisania pełnego adresu, a ogólna struktura adresu rezerwuje na ten cel 14 bajtów. Stąd wynika istnienie końcowego pola struktury, które uzupełnia strukturę *sockaddr_in* do rozmiaru ogólnej struktury *sockaddr*.

3.3. Adresowanie gniazd w dziedzinie Unix'a

Ten format adresu jest używany w odniesieniu do gniazd, które znajdują się lokalnie w hoście użytkownika (na przykład w komputerze klasy PC z zainstalowanym systemem LINUX). Adres lokalny określony jest za pomocą gniazdowej struktury adresowej dla dziedziny UNIX'a, która nosi nazwę

sockaddr_un.

```
#include <sys/un.h>

struct sockaddr_un
{
  sa_family_t sun_family // Tutaj należy wpisać domenę AF_UNIX //
  char sun_path[]; // tutaj wpisujemy ścieżkę pliku //
}
```

Strukturę powinno się wypełniać w następujący sposób :

- W polu “sun_family” określającego rodzinę adresów należy wpisać wartość AF_UNIX lub AF_LOCAL. Tak wpisana wartość oznacza, że adresy będą formowane zgodnie z lokalnymi (UNIX'owymi) zasadami.

- W polu “sun_path[]” wpisujemy UNIX'ową nazwę ścieżki do pliku. Należy pamiętać, że adresami protokołowymi używanymi do identyfikowania klientów i serwerów w dziedzinie UNIX'a są nazwy ścieżkowe w obrębie zwykłego systemu plików. Jednak nie są to zwykłe pliki – nie można w zwyczajny sposób ani pobrać danych z tych plików, ani odsyłać do nich danych. Może to robić tylko program, który powiązał taką nazwę ścieżkową (tak jakby adres gniazda) z gniazdem w dziedzinie UNIX'a.

Przykład : adresowanie gniazd w dziedzinie UNIX'a
źródło : bind.c

```
#include <sys/types.h>
#include <sys/socket.h>
#include <stdio.h>
#include <sys/un.h>
#include <unistd.h>

void blad()
{
  perror("Blad programu.");
  exit(1);
}

int main()
{
  int gniazdo;
  int dlugosc;
  int test1,test2;
  char *nazwa;
  struct sockaddr_un adres;
```

```
 // Ustalenie nazwy nowego gniazda
 nazwa="gniazdo";

 //Usuniecie starego gniazda
 unlink(nazwa);

 //Utworzenie gniazda
 test1=gniazdo=socket(AF_UNIX, SOCK_STREAM,0);
 if (test1==-1) blad();

 //Zwiazanie gniazda z adresem
 adres.sun_family=AF_UNIX;
 strcpy(adres.sun_path,nazwa);
 dlugosc=sizeof(adres);
 test2=bind(gniazdo,(struct sockaddr *)&adres,dlugosc);
 if (test2==-1) blad();

    printf("Gniazdo o deskryptorze %d zostalo zwiazane z adresem:\nNazwa pliku :
%s\n",test1,nazwa);
 exit(0);
}
```

3.4. Problemy z adresowaniem

Gniazdowe struktury adresowe są zawsze przekazywane przez odniesienie, gdy są argumentem jakieś funkcji gniazd. Ale funkcje gniazd, które pobierają jako argument jeden z tych wskaźników, muszą działać na gniazdowych strukturach adresowych właściwych dla dowolnej rodziny protokołów obsługiwanej przez system operacyjny. Dlatego w każdym wywołaniu tych funkcji trzeba rzutować wskaźnik do właściwej dla danego protokołu gniazdowej struktury adresowej, tak aby otrzymać wskaźnik do ogólnej gniazdowej struktury adresowej; na przykład :

struct sockaddr_in serv; // gniazdowa struktura adresowa IP

// zapełniamy tę strukturę //

*bind(sockfd, (struct sockaddr *)&serv, sizeof(serv));*

Jeżeli opuścimy rzutowanie *(struct sockaddr *)*, to kompilator języka C utworzy ostrzeżenie informujące o tym, że do drugiego argumentu funkcji *bind()* podstawiono niezgodny typ wskaźnika.

3.5. Porządek podawania adresów

Różne procesory w różny sposób grupują bajty danych, tworząc liczby całkowite składające się z 16, 32 lub większej liczby bitów. Rozważmy liczbę 16-bitową, która składa się z dwóch bajtów. Istnieją dwa sposoby umieszczania tych dwóch bajtów w pamięci : albo bajt mniej znaczący otrzyma wcześniejszy adres i wtedy mówimy, że przyjęto **kolejność z pierwszeństwem bajtu mniej znaczącego** *(ang. little-endian)*, albo bajt bardziej znaczący otrzyma wcześniejszy adres i wtedy mówimy, że przyjęto **kolejność z pierwszeństwem bajtu bardziej znaczącego** (ang. *big-endian)*. W praktyce będziemy rozróżniać dwa uporządkowania kolejności bajtów :

- Porządek hosta będący porządkiem właściwym dla danego procesora. Na przykład procesory firmy Intel stosują kolejność bajtów z pierwszeństwem bajtu mniej znaczącego, a procesor Motorola 6800 używa kolejności bajtów z pierwszeństwem bajtu bardziej znaczącego.

- Porządek sieciowy będący uporządkowaniem bajtów z pierwszeństwem bajtu bardziej znaczącego.

Kolejność z pierwszeństwem bajtu mniej znaczącego : IBM 370, Motorola 68000, Pyramid
Kolejność z pierwszeństwem bajtu bardziej znaczącego : Intel 80x86 (IBM PC), DEC VAX

Tabela 3.1. Kolejność bajtów przyjęta dla różnych komputerów i procesorów

Programiści piszący oprogramowanie sieciowe muszą rozważać przekształcenia między systemową (porządek hosta), a sieciową (porządek sieciowy) kolejnością bajtów. Przekształceń tych dokonują cztery poniższe funkcje : *htons, htonl, ntohs* oraz ntohl :

```
#include <netinet/in.h>

uint16_t htons(uint16t host16bitvalue);
uint32_t htonl(uint32t host32bitvalue);
uint16_t ntohs(uint16t net16bitvalue);
uint32_t ntohl(uint32t net32bitvalue);
```

W nazwach tych funkcji użyto liter : *h* na oznaczenie stacji (ang. *host*), n - sieci (ang. *net*), s - typu *short int* oraz l – typu *long int*. Pierwsze dwie funkcje dokonują przekształcenia z porządku hosta na porządek sieciowy. Argumentami tych funkcji są odpowiednio 16-bitowa liczba całkowita bez znaku i 32-bitowa liczba całkowita bez znaku. Natomiast kolejne dwie funkcje dokonują przekształcenia z porządku sieciowego na porządek hosta.

3.6. Funkcje konwertowania adresów

Adresy internetowe są zazwyczaj zapisywane jako czwórki liczb dziesiętnych rozdzielonych kropkami. Każda liczba odpowiada jednemu bajtowi 32-bitowego adresu. Na przykład liczba w zapisie szesnastkowym 0x0102FF04 oznacza adres internetowy 1.2.255.4. Nierzadko programista aplikacji sieciowych potrzebuje możliwości konwertowania adresu z kropkowej notacji czwórkowej na jego 32-bitową wartość w kodzie dwójkowym, uporządkowaną zgodnie z sieciową kolejnością bajtów lub przekształcenia odwrotnego. Służą do tego trzy funkcje *inet_aton()*, *inet_addr()*, *inet_ntoa()*.

```
#include <arpa/inet.h>

int inet_aton(const char *strptr, struct in_addr *addrptr);

in_addr_t inet_addr(const char *strptr);

char *inet_ntoa(struct in_addr inadr);
```

Pierwsza z tych funkcji *inet_aton()*, przekształca napis w języku C wskazany przez argument *strptr* na 32-bitową liczbę całkowitą bez znaku zapisaną w sieciowej kolejności bajtów. Wynik przekształcenia będzie zapamiętany pod adresem wskazywanym przez drugi argument funkcji *addrptr*. Funkcja przekazuje 1, jeśli napis jest poprawny oraz 0 w przeciwnym wypadku.

Funkcja *inet_addr()* wykonuje to samo przekształcenie jak funkcja *inet_aton()*. Nie umieszcza jednak wyniku w strukturze, tylko zwraca 32-bitową liczbę całkowitą bez znaku w sieciowej kolejności bajtów.

Funkcja *inet_ntoa()* przekształca adres IP w postaci 32-bitowej liczby całkowitej bez znaku zapisanej w sieciowej kolejności bajtów na odpowiedni napis w kropkowej notacji czwórkowej.

3.7. Funkcja gethostbyname()

Adresami liczbowymi zajmują się jednak komputery. Dla ludzi zostały stworzone łatwiejsze do zapamiętanie adresy literowe. Na przykład adres serwera firmy Microsoft to *http://microsoft.com*. Zauważmy, że o wiele trudniej byłoby zapamiętać jego numer IP, który jest postaci *207.46.130.149*. Tą zamianą zajmuje się usługa nazewnictwa domen DNS (ang. *Domain Name Service*), która przyporządkowuje unikatowym adresom IP nazwy. Znając nazwę komputera, możemy znaleźć jego adres IP, wywołując funkcję *gethostbyname()* operującą na bazie danych hostów.

```
#include <netdb.h>

struct hostent *gethostbyname(const char *name);
```

Funkcja sprawdza plik konfiguracyjny sieci */etc/hosts*, bądź też łączy się z sieciowymi usługami informacyjnymi, takimi jak NIS (ang. *Network Information Service*) lub DNS (ang. *Domain name Service*). W przypadku pomyślnego wykonania wywołanie *gethostbyname()* zwróci wskaźnik do struktury *struct hostent*, której pole *h_addr_list* zawiera numer IP danego hosta.

```
struct hostent
{
char *h_name;     /* nazwa hosta */
char **h_aliases;     /* lista aliasów (pseudonimów) */
int h_addrtype;     /* typ adresu */
int h_length;     /* długość adresu w bajtach */
char **h_addr_list;     /* lista adresów (porządek sieciowy) */
}
```

W przeciwnym wypadku funkcja zwróci wskaźnik pusty. Funkcję tę ilustruje poniższy przykład :

Przykład : uzyskanie informacji o naszym komputerze
źródło : ghs.c

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <stdio.h>
#include <unistd.h>

int main()
{
  char *host;
  char **aliasy;
  int adres_typ;
  int adres_dlugosc;
  char **adres_lista;
  struct hostent *hostinfo;

  //Ustalenie nazwy komputera
  host="localhost";

  hostinfo=gethostbyname(host);
  if (!hostinfo)
```

```
  {
   perror("Blad programu.");
   exit(1);
  }

 printf("Wyniki dla komputera : %s\n",host);

 //Nazwa komputera
 printf("Nazwa komputera : %s\n",hostinfo->h_name);

 //Lista aliasow dla komputera
 aliasy=hostinfo->h_aliases;
 while (*aliasy)
  {
  printf("Alias dla komputera : %s\n",*aliasy);
  aliasy++;
  }

 //Lista adresow IP dla komputera
 adres_lista=hostinfo->h_addr_list;
 printf("Adres IP komputera : ");
 while (*adres_lista)
  {
  printf("%s\n",inet_ntoa(*(struct in_addr *)*adres_lista));
  adres_lista++;
  }

 //Typ adresu
 adres_typ=hostinfo->h_addrtype;
 printf("Typ adresu : %d\n",adres_typ);

 //Dlugosc adresu w bajtach
 adres_dlugosc=hostinfo->h_length;
 printf("Dlugosc adresu w bajtach : %d\n",adres_dlugosc);

 exit(0);
}
```

Powyższy program sprawdził w bazach danych informacje na temat naszego komputera, wywołując funkcję *gethostbyname()*. Zauważmy, że lista adresów wypisująca adres IP naszego komputera wymaga rzutowania na określony typ adresu i musi zostać zamieniona z porządku sieciowego do nadającego się do wyświetlenia ciągu znaków. W tym momencie wykorzystałem wyżej omówioną funkcję *inet_ntoa()*.

Rozdział 4. Podstawowe funkcje gniazd

W tym rozdziale opiszę podstawowe funkcje wchodzące w skład interfejsu gniazd BSD.

4.1. Tworzenie nowego gniazda

Nowe gniazdo można utworzyć za pomocą funkcji systemowej *socket()*.

```
#include <sys/types.h>
#include <sys/socket.h>

int socket(int domain, int type, int protocol);
```

Pierwszy argument *domain* określa rodzinę protokołów i jest jedną ze stałych wymienionych w tabeli 4.1. Parametr *type* definiuje typ gniazda i jest jedną ze stałych przedstawionych w tabeli 4.2. Nie wszystkie pary argumentów *domain* i *type* są poprawne. Pary poprawne i wybrany dla danej pary protokół przedstawia tabela 4.3. Słowo “tak” w klatce oznacza, że dana para jest poprawna, ale nie istnieje dla niej poprawny akronim.

Dziedzina gniazda	***Opis***
AF_UNIX lub AF_LOCAL	Protokoły dziedziny UNIX'a
AF_INET	Protokoły rodziny TCP/IP

Tabela 4.1. Rodziny protokołów dla funkcji socket().

Typ gniazda	***Opis***
SOCK_STREAM	Gniazdo strumieniowe
SOCK_DGRAM	Gniazdo datagramowe

Tabela 4.2. Typy gniazd dla funkcji socket()

	AF_UNIX (AF_LOCAL)	***AF_INET***
SOCK_STREAM	tak	TCP
SOCK_DGRAM	tak	UDP

Tabela 4.3. Pary argumentów domain i type dla funkcji socket().

Mogłoby się wydawać, że po zdefiniowaniu rodziny protokołów i typu gniazda nie ma potrzeby definiowania niczego więcej. Jednak mimo iż dla danej rodziny protokołów i typu gniazda jest tylko jeden protokół, są sytuacje, gdy tych

protokołów jest więcej. Parametr *protocol* pozwala w takiej sytuacji wybrać jeden z dostępnych protokołów. W praktyce programista powinien ustawić argument *protocol* na wartość 0, co spowoduje, że jądro systemu wybierze domyślny protokół.

Zestawienie parametrów funkcji *socket()* i ich wpływ na sposób komunikacji przedstawia tabela 4.4.

Domena	***Typ gniazda***	***Opis***
AF_UNIX	SOCK_STREAM	Dostarcza gniazdo strumieniowe do komunikacji w obrębie lokalnego hosta. Usługa jest połączeniowa i bezpieczna.
AF_UNIX	SOCK_DGRAM	Dostarcza gniazdo datagramowe do komunikacji w obrębie lokalnego hosta. Usługa jest bezpołączeniowa i zazwyczaj bezpieczna.
AF_INET	SOCK_STREAM	Stosowane dla potrzeb przesyłania danych przez Internet między dwoma połączonymi gniazdami. Używa się tutaj protokołu TCP, który jest niezawodny.
AF_INET	SOCK_DGRAM	Służy do przesyłania danych przez Internet między dwoma niepołączeniowymi gniazdami. Używa się tutaj protokołu UDP, który nie gwarantuje bezpieczeństwa przesyłania danych.

Tabela 4.4. Zestawienie parametrów funkcji socket() związanych z komunikacją lokalną i internetową.

Jeżeli powiedzie się wykonanie funkcji *socket()* to przekaże ona małą nieujemną liczbę całkowitą, która jest deskryptorem gniazda (ang. *socket descriptor*) i służy jako element identyfikujący to gniazdo we wszystkich następnych wywołaniach funkcji systemowych (np. dla funkcji *connect()*, którą opiszę później). W przypadku błędu funkcja zwróci wartość -1 i ustawi zmienną *errno* na wartość całkowitą dodatnią wskazującą na rodzaj błędu. Wybrane stałe opisujące rodzaj błędu przedstawia tabela 4.5.

Errno	*Opis*
EINVAL	Błędne dane
EPROTONOSUPPORT	Wartość typu i protokołu nie są dozwolone w podanej dziedzinie gniazda
EMFILE	Przepełniona tablica deskryptorów procesu
ENFILE	Przepełniona tablica plików
EACCES	Brak uprawnień do utworzenia gniazda o podanych parametrach.
ENOSR	Brak zasobów systemowych
ENOBUFS	Brak pamięci na bufory gniazda

Tabela 4.5. Wybrane stałe opisujące rodzaj błędu dla funkcji socket().

Utworzenie nowego gniazda za pomocą funkcji *socket()*

Aby utworzyć gniazdo internetowe korzystające z protokołu TCP, należy wywołać funkcję systemową socket() z dziedziną AF_INET oraz typem gniazda SOCK_STREAM.

```
sockfd=socket(AF_INET, SOCK_STREAM,0)
/* tworzy gniazdo korzystające z protokołu TCP */
```

Analogicznie można utworzyć gniazdo internetowe korzystające z protokołu UDP, zmieniając typ gniazda z SOCK_STREAM na SOCK_DGRAM.

```
sockfd=socket(AF_INET, SOCK_DGRAM,0)
/* tworzy gniazdo korzystające z protokołu UDP */
```

4.2. Związanie gniazda z adresem

Gniazdo utworzone za pomocą funkcji *socket()* jest anonimowe. Aby związać to gniazdo z danym adresem protokołowym należy zastosować wywołanie systemowe *bind()*.

```
#include <sys/types.h>
#include <sys/socket.h>

int bind(int socket, const struct sockaddr *adress, socklen_t adress_len)
```

Funkcja *bind()* przypisuje adres określony parametrem *adress* do nienazwanego gniazda o deskryptorze *socket*. Długość struktury adresowej *adress* jest przekazywana w argumencie *adress_len*. Dla gniazd internetowych

adres składa się z 32-bitowego adresu IP oraz 16-bitowego numeru portu TCP lub UDP. Natomiast dla gniazd lokalnych adresem jest po prostu nazwa pliku. Wywołując funkcję *bind()* dla gniazda internetowego możemy określić : numer portu, adres IP, albo obie te wielkości, albo też żadnej nie podawać.

Adres IP	***Numer portu***	***Wynik funkcji bind()***
Brak	Brak	Jądro systemu wybiera adres IP oraz numer portu.
Brak	Podany	Jądro wybiera adres IP, proces określa numer portu.
Podany	Brak	Proces określa adres IP, jądro wybiera numer portu.
Podany	Podany	Proces określa adres IP oraz numer portu.

Tabela 4.6. Sposób działania funkcji bind() w zależności od określenia adresu IP oraz numeru portu.

Jeżeli wywołując funkcję *bind()* nie określimy numeru portu, to jądro systemu wybierze port efemeryczny dla danego gniazda, jeśli będzie wywołana funkcja *connect() (*dla gniazda klienta) lub *listen()* (dla gniazda serwera). W przypadku nieokreślenia adresu IP, jądro systemu wybierze adres źródłowy IP wtedy, kiedy gniazdo będzie połączone. Jednak procesy pełniące rolę serwerów nie powinny wyrażać zgody, aby jądro systemu dokonywało za nich wyboru, ponieważ są one zazwyczaj rozpoznawane na podstawie ich ogólnie znanych numerów portów.

Errno	***Opis***
EBADF	Błędny deskryptor
ENOTSOCK	Deskryptor nie odnosi się do gniazda
EINVAL	Deskryptor odnosi się do już nazwanego gniazda
EADDRNOTAVAIL	Adres jest niedostępny
EADDRINUSE	Adres ma już przypisane gniazdo
EACCESS	Nie można utworzyć nazwy w systemie plików ze względu na brak zezwolenia (dotyczy gniazd AF_UNIX)
ENOTDIR	Niefortunnie wybrana nazwa pliku (dotyczy gniazd AF_UNIX)
ENAMETOOLONG	Niefortunnie wybrana nazwa pliku (dotyczy gniazd AF_UNIX)

Tabela 4.7. Wybrane stałe opisujące rodzaj błędu dla funkcji bind().

W przypadku pomyślnego wykonania funkcja zwraca 0. W przypadku błędu, zwraca -1 i ustawia *errno* na wartość całkowitą dodatnią wskazującą na rodzaj błędu.

Proces może dowiązać konkretny adres IP do swojego gniazda jeżeli ten adres należy do interfejsu danej stacji. W wypadku procesu klienta dowiązany konkretny adres IP staje się adresem źródłowym IP, którego używa się dla datagramów IP wysyłanych z tego gniazda. W przypadku procesu serwera takie dowiązanie powoduje, że gniazdo może przyjmować tylko te nadchodzące od klientów połączenia, które są przeznaczone dla tego adresu IP. Aby umożliwić działanie serwerów na komputerach wielosieciowych, interfejs gniazd zawiera specjalną stałą symboliczną, **INADDR_ANY**, która pozwala, aby serwer korzystał z danego portu na każdym z adresów IP danego komputera.

Przykład : związanie gniazda z adresem internetowym za pomocą funkcji bind()
źródło : bind2.c

```
#include <sys/types.h>
#include <sys/socket.h>
#include <stdio.h>
#include <sys/un.h>
#include <unistd.h>
#include <netinet/in.h>
#include <arpa/inet.h>

void blad()
{
  perror("Blad programu.");
  exit(1);
}

int main()
{
  int gniazdo;
  int dlugosc;
  int test1,test2;
  int port;
  char *adresip;
  struct sockaddr_in adres;

  //Utworzenie gniazda
  test1=gniazdo=socket(AF_INET, SOCK_STREAM,0);
  if (test1==-1) blad();

  //Ustalenie nazwy (adresu) dla gniazda
  adresip="127.0.0.1";
  port=9734;
```

```
  //Zwiazanie gniazda z adresem
  adres.sin_family=AF_INET;
  adres.sin_addr.s_addr=inet_addr(adresip);
  adres.sin_port=port;
  dlugosc=sizeof(adres);

  test2=bind(gniazdo,(struct sockaddr *)&adres,dlugosc);
  if (test2==-1) blad();

  printf("Gniazdo o deskryptorze plikow %d zostalo zwiazane z adresem :\nAdres IP :%s
        , Numer portu : %d\n",gniazdo,adresip,port);
}
```

4.3. Tworzenie kolejki na gnieździe

Wywołując funkcję *listen()*, przekształcamy gniazdo połączeniowe serwera utworzone za pomocą funkcji *socket()* w gniazdo nasłuchujące, w którym przychodzące od klientów połączenia będą akceptowane przez jądro systemu. Te następujące po sobie wywołania trzech funkcji : *socket(), bind()* i *listen()* to trzy czynności, które trzeba wykonać dla każdego serwera działającego zgodnie z protokołem TCP.

```
#include <sys/types.h>
#include <sys/socket.h>

int listen(int socket, int backlog)
```

Pierwszym argumentem funkcji jest deskryptor gniazda serwera, natomiast używając parametru *backlog* możemy ograniczyć maksymalną liczbę oczekujących na obsłużenie klientów, którzy mogą być przetrzymywani w kolejce. Próby połączenia przekraczające tę liczbę będą odrzucane przez jądro systemu. Dawniej w przykładowych program używano zawsze liczby pięć jako argumentu *backlog*, ponieważ była to największa jego wartość dopuszczana w systemie 4.2BSD. Było to dobre rozwiązanie w latach osiemdziesiątych, kiedy serwery obsługiwały tylko kilkaset połączeń dziennie. Dzisiaj kiedy obciążone serwery mogą obsługiwać nawet kilka milionów połączeń dziennie, tak mała wartość parametru *backlog* okazuje się zupełnie nieodpowiednia.

Funkcja *listen()* zwraca 0 w przypadku pomyślnego wykonania. W przypadku błędu funkcja zwraca wartość -1, a jego przyczynę będzie można ustalić poprzez zbadanie zmiennej *errno*.

Errno	*Opis*
EBADF	Argument *socket* nie jest poprawnym deskryptorem pliku.
ENOTSOCK	Argument *socket* nie jest deskryptorem gniazda.
EOPNOTSUPP	Gniazdo o deskryptorze *socket* nie zezwala na wykonanie tej operacji.

Tabela 4.8. Wybrane stałe opisujące rodzaj błędu dla funkcji listen().

4.4. Akceptowanie połączeń

Serwer używający gniazda połączeniowego wywołuje funkcję *accept()*, aby z wierzchu kolejki utworzonej przez funkcję *listen()* pobrała następne oczekujące połączenie. Jeżeli kolejka jest pusta, to proces serwera zostanie uśpiony.

```
#include <sys/types.h>
#include <sys/socket.h>

int accept(int socket, struct sockaddr *adress, socklen_t adress_len)
```

Pierwszym argumentem jest jak zwykle deskryptor gniazda serwera utworzonego przez funkcję *socket()*. Adres wywołującego klienta zostanie umieszczony w strukturze *sockaddr*, na którą wskazuje argument *adress*. Długość tej struktury znajduje się w parametrze *adress_len*. Jeżeli adres klienta będzie dłuższy niż ta wartość, zostanie on obcięty. Jeżeli podczas wykonywania funkcji *accept()* nie wystąpi błąd, to wartość przekazana przez funkcję będzie deskryptorem nowego gniazda, automatycznie utworzonego przez jądro systemu i mającego ten sam typ co gniazdo utworzone przez funkcję *socket()*. Omawiając funkcję *accept()* mamy do czynienia z dwoma gniazdami. Pierwszy argument funkcji jest deskryptorem pierwszego gniazda, zwanego **gniazdem nasłuchującym** (ang. *listening socket)*, utworzonego przez funkcje *socket()* i następnie użytego w wywołaniach systemowych *bind()* oraz *listen()*. Natomiast wartością przekazaną przez funkcję *accept()* jest deskryptor gniazda zwanego **gniazdem połączonym** (ang. *connected socket*).

To rozróżnienie gniazd jest bardzo ważne. Serwer zazwyczaj tworzy tylko jedno gniazdo nasłuchujące, które istnieje przez cały czas jego działania. Wywołanie *accept()* powoduje utworzenie gniazda połączonego dla każdego połączenia z klientem, które zostało zaakceptowane. Celem takiego gniazda jest tylko komunikacja z danym klientem. Kiedy serwer zakończy obsługę tego klienta, wtedy połączone gniazdo zostanie zamknięte.

4.5. Żądanie nawiązania połączenia

Klient korzystający z gniazda połączeniowego używa funkcji *connect()* do ustanowienia połączenia z serwerem.

```
#include <sys/types.h>
#include <sys/socket.h>

int connect(int socket, const struct sockaddr *adress, socklen_t adress_len)
```

Argument *socket* oznacza deskryptor gniazda, który przekazuje funkcja *socket().* Drugi i trzeci argument to wskaźnik do gniazdowej struktury adresowej i jej rozmiar. W gniazdowej strukturze adresowej musi znajdować się adres serwera. Jeżeli komunikacja odbywa się w środowisku Internetu to musimy podać adres IP oraz numer portu serwera, jeżeli komunikacja odbywa się wewnątrz jednego systemu operacyjnego to podajemy nazwę pliku związaną z gniazdem serwera. Przed wywołaniem funkcji *connect()* klient nie musi wywoływać funkcji *bind()*, ponieważ w razie potrzeby jądro systemu samo wybierze dla niego port efemeryczny i źródłowy adres IP. Powrót z funkcji nastąpi tylko wtedy, kiedy będzie ustanowione połączenie z serwerem albo pojawi się błąd. Jeżeli połączenie klienta z serwerem nie może być od razu zrealizowane, funkcja *connect()* zablokuje się na jakiś nieokreślony czas. Po upłynięciu limitu czasu połączenie zostanie zaniechane i funkcja zwróci błąd.

Wartością zwracaną przez funkcję *connect()* jest zero w przypadku pomyślnego wykonania oraz -1 w przeciwnym przypadku. W razie błędu zmienna *errno* przyjmie dodatnią wartość całkowitą.

Errno	***Opis***
EBADF	W parametrze *socket* określono niepoprawny deskryptor.
EALREADY	Gniazdo nawiązało już połączenie.
ECONNREFUSED	Serwer odrzucił próbę połączenia.
EINTR	Wywołanie *connect* zostało przerwane przez sygnał.

Tabela 4.9. Wybrane stałe opisujące rodzaj błędu dla funkcji connect().

4.6. Funkcje systemowe read() i write()

Działanie tych funkcji na deskryptorze gniazda jest analogicznie do ich działania na deskryptorach plików. Funkcje te służą do wymiany danych między dwoma gniazdami już połączonymi (transmisja połączeniowa).

```
#include <unistd.h>

int read(int sockfd, char *buf, int nbytes)
int write(int sockfd, char *buf, int nbytes)
```

Parametr *sockfd* określa deskryptor gniazda, utworzony przez funkcję *socket()* lub *accept()*. Argument *buf* jest wskaźnikiem do bufora, zawierającego dane do wysłania przez funkcje *write()*, lub pod którym zostaną umieszczone dane do odebrania przez funkcję *read()*. Parametr *nbytes* definiuje liczbę bajtów do wysłania przez funkcję *write()* lub maksymalną liczbę bajtów, którą można jednorazowo odebrać, wywołując funkcję *read()*.

Funkcja zwraca liczbę przeczytanych lub zapisanych bajtów w przypadku sukcesu lub -1 w razie błędu. Liczba odebranych/wysłanych bajtów może być mniejsza niż wartość *nbytes*.

4.7. Funkcje systemowe sendto() i recvfrom()

Funkcje *sendto()* i *recvfrom()* są używane do komunikacji procesów klienta i serwera przy użyciu gniazd bezpołączeniowych. Klient nie ustanawia połączenia z serwerem. W zamian klient po prostu wysyła datagram do serwera, korzystając z funkcji *sendto()*, która wymaga podanie adresu docelowego (adresu serwera) jako argumentu wywołania. Podobnie też serwer nie akceptuje połączenia z klientem. W zamian serwer po prostu wywołuje funkcję *recvfrom()*, która czeka, aż nadejdą dane od jakiegoś klienta. Funkcja *recvfrom()* przekazuje adres protokołowy klienta razem z datagramem, więc serwer może wysłać odpowiedź do właściwego klienta. W sensie konstrukcyjnym, powyższe funkcje są podobne do funkcji *read()* i *write()*, ale mają trzy dodatkowe argumenty.

```
#include <sys/types.h>
#include <sys/socket.h>

int sendto(int sockfd,char *buff, int nbytes, int flags, struct sockaddr *to, int addrlen);
int recvfrom(int sockfd,char *buff, int nbytes, int flags, struct sockaddr *from, int *addrlen);
```

Pierwsze trzy argumenty *sockfd*, *buff* oraz *nbytes* są takie same jak pierwsze trzy argumenty funkcji *read()* oraz write() i oznaczają odpowiednio : deskryptor, wskaźnik do bufora, do którego się pobiera lub z którego się odsyła dane, oraz liczbę pobranych lub odesłanych bajtów. Argument *flags* określający sygnalizatory jest przeważnie równy 0. Argument *to* funkcji *send()* wskazuje na gniazdową strukturę adresową, zawierającą adres protokołowy (adres IP oraz numer portu dla gniazd internetowych lub nazwę pliku dla gniazd lokalnych), pod który mają być wysłane dane. Rozmiar tej struktury jest określony przez

argument *addrlen*. Funkcja *recvfrom()* umieszcza adres protokołowy nadawcy datagramu w gniazdowej strukturze adresowej wskazywanej przez argument *from*. Liczba bajtów zapamiętanych w tej strukturze będzie również przekazana do programu, który wywołał funkcję *recvfrom()*, jako liczba całkowita wskazywana przez argument *addrlen*. Zauważmy, że ostatni argument przekazywany do funkcji *sendto()* jest liczbą całkowitą, podczas gdy ostatni argument funkcji *recvfrom()* jest wskaźnikiem do wartości całkowitej. Dzieje się tak, ponieważ ostatni argument funkcji *recvfrom()* służy również do przekazywania dodatkowego wyniku tej funkcji. Obie funkcje jako swoją wartość przekazują rozmiar danych lub odesłanych.

4.8. Zamykanie gniazda

Możemy zamknąć połączenie od strony klienta lub serwera, używając funkcji *close()*.

```
#include <unistd.h>

int close(int socket)
```

Funkcja zawiera tylko jeden parametr będący deskryptorem zamykanego gniazda. Jeżeli zamykane gniazdo jest związane z protokołem zapewniającym niezawodne doręczenie danych (na przykład protokół TCP), to system operacyjny musi zadbać o to, aby wysłać wszystkie dane, które pozostają wewnątrz jądra, a muszą być przesłane. Zazwyczaj następuje natychmiastowy powrót z funkcji systemowej *close()* do systemu, ale jądro próbuje jeszcze wysłać wszystkie dane znajdujące się w kolejce. Funkcja zwróci wartość zero, jeżeli gniazdo zostanie zamknięte z powodzeniem.

Większe możliwości daje funkcja *shutdown()*, która pozwala zlikwidować połączenie całkowicie lub częściowo.

```
#include <sys/socket.h>

Int shutdown(int socket, int how)
```

Argument *socket* oznacza deskryptor gniazda, a parametr *how* określa sposób likwidacji połączenia. Dopuszczalne są następujące wartości argumentu *how* :

- *SHUT_RD* – następuje zamknięcie części czytającej połączenia. Nie można odbierać z gniazda żadnym nowych danych, a wszystkie dane obecnie znajdujące się w buforze odbiorczym są odrzucane.

- *SHUT_WR* – z danego gniazda nie można już wysyłać żadnych danych.

- *SHUT_RDWR* – dane gniazdo nie może ani pobierać, ani wysyłać danych. Równoważne działanie można uzyskać, wywołując funkcję *shutdown()* dwukrotnie : najpierw z argumentem *SHUT_RD*, a potem z argumentem *SHUT_WR.*

4.9. Przykłady

W poprzednich podrozdziałach omówiłem interfejs programowy gniazd, uwzględniając poszczególne funkcje, ich parametry i spełniane przez nie operacje na gniazdach. Teraz będę dalej omawiał ten interfejs, analizując przykładowe programy klienta i serwera komunikujące się przy pomocy gniazd. Dla zmniejszenia rozmiaru przykładowych programów i uwypuklenia odwołań do funkcji interfejsu gniazdowego, zdecydowałem się na implementację bardzo prostej usługi :

- Proces klienta wysyła do procesu serwera znak 'a'.
- Proces serwera odbiera od procesu klienta wysłany znak.
- Proces serwera zmienia znak 'a' na znak 'b'.
- Proces serwera odsyła procesowi klienta zmieniony znak.
- Proces klienta otrzymuje zmieniony znak.

Wszystkie przedstawione poniżej programy będą wykonywać te same zadanie wykorzystując różne typy gniazd (strumieniowe, datagramowe) i działając w różnych środowiskach (system LINUX, Internet).

Pierwszy przykład opisuje dwa procesy komunikujące się ze sobą za pomocą gniazd połączeniowych. Środowiskiem gniazd jest system LINUX.

Przykład : system komunikacyjny oparty na modelu klient-serwer.
Środowisko : LINUX
Typ gniazd : Strumieniowe

Pierwszy program : serwer
źródło : s.c

```
#include <sys/types.h>
#include <sys/socket.h>
#include <stdio.h>
#include <sys/un.h>
#include <unistd.h>

void blad()
{
  perror("Blad serwera.");
  exit(1);
}
```

```
int main()
{
  int gniazdo_serwera;
  int gniazdo_dla_klienta;
  int serwer_dlugosc;
  int klient_dlugosc;
  int test1,test2,test3,test4,test5;
  char znak;
  struct sockaddr_un adres_serwera;
  struct sockaddr_un adres_klienta;

  //Usuniecie starego gniazda
  unlink("gniazdo_serwera");

  //Utworzenie gniazda serwera
  test1=gniazdo_serwera=socket(AF_UNIX, SOCK_STREAM,0);
  if (test1==-1) blad();

  //Zwiazanie gniazda serwera z adresem
  adres_serwera.sun_family=AF_UNIX;
  strcpy(adres_serwera.sun_path,"gniazdo_serwera");
  serwer_dlugosc=sizeof(adres_serwera);
  test2=bind(gniazdo_serwera,(struct sockaddr *)&adres_serwera,serwer_dlugosc);
  if (test2==-1) blad();

  //Utworzenie kolejki dla oczekujacych polaczen
  test3=listen(gniazdo_serwera,5);
  if (test3==-1) blad();

  printf("Serwer nasluchuje...\n");

  //Komunikacja gniazda serwera z gniazdem klienta
  while(1)
    {
    //Przyjęcie polaczenia przez gniazdo serwera
    gniazdo_dla_klienta=accept(gniazdo_serwera,(struct
                               sockaddr *)&adres_klienta,&klient_dlugosc);
    test4=read(gniazdo_dla_klienta,&znak,1);
    if (test4>0) printf("Odebralem do klienta znak %c.\n",znak);
    else blad();

    znak='b';

    test5=write(gniazdo_dla_klienta,&znak,1);
    if (test5>0) printf("Wysylam dane do klienta znak %c.\n",znak);
    else blad();

    close(gniazdo_dla_klienta);
    }
  close(gniazdo_serwera);
```

```
  exit(0);
}
```

Drugi program : klient
źródło : k.c

```
#include <sys/types.h>
#include <sys/socket.h>
#include <stdio.h>
#include <sys/un.h>
#include <unistd.h>


void blad()
{
  perror("Blad klienta.");
  exit(1);

}

int main()
{
  int gniazdo_klienta;
  int serwer_dlugosc;
  char znak;
  int test1,test2,test3,test4,test5;
  struct sockaddr_un adres_serwera;

  //Utworzenie gniazda klienta
  test1=gniazdo_klienta=socket(AF_UNIX, SOCK_STREAM,0);
  if (test1==-1) blad();

  //Ustalenie adresu serwera
  adres_serwera.sun_family=AF_UNIX;
  strcpy(adres_serwera.sun_path,"gniazdo_serwera");
  serwer_dlugosc=sizeof(adres_serwera);
    //Polaczenie gniazda klienta z gniazdem serwera
  test1=connect(gniazdo_klienta,(struct sockaddr *)&adres_serwera,serwer_dlugosc);
  if (test1==-1) blad();
  znak='a';
  //Komunikacja gniazda serwera z gniazdem klienta
  test4=write(gniazdo_klienta,&znak,1);
  if (test4>0) printf("Wyslalem do serwera znak %c.\n",znak);
  else blad();

  test5=read(gniazdo_klienta,&znak,1);
  if (test5>0) printf("Otrzymalem do serwera znak %c.\n",znak);
  else blad();
  close(gniazdo_klienta);
  exit(0);
```

}

- **Opis powyższego programu : od strony serwera.**

- Za pomocą funkcji *socket()* tworzę lokalne (stała AF_UNIX) gniazdo strumieniowe (stała SOCK_STREAM), czyli gniazdo działające w obrębie jednego systemu operacyjnego. Funkcja ta przekazuje małą liczbę całkowitą, która służy jako deskryptor identyfikujący to gniazdo w następnych wywołaniach systemowych.

- W gniazdowej strukturze adresowej dla dziedziny UNIX'a (struktura *sockaddr_un* określona identyfikatorem *adres_serwera*) umieszczam adres serwera, odpowiednio wypełniając pola wchodzące w skład tej struktury. W polu *sun_family* wpisuję wartość *AF_UNIX*, co spowoduje, że adres będzie formowany zgodnie z lokalnymi (UNIX'owymi) zasadami adresowania. Natomiast pole *sun_path* wypełniam nazwą ścieżki do pliku będącej adresem gniazda serwera (w tym przypadku wpisuję wartość *gniazdo_serwera*). Po poprawnym zdefiniowaniu struktury adresowej wiążę gniazdo utworzone za pomocą funkcji *socket()* z adresem zdefiniowanym w tej strukturze. Dokonuję tego używając funkcji systemowej *bind()*.

- Wywołując funkcje *listen()*, przekształcam gniazdo serwera w gniazdo nasłuchujące, w którym przychodzące od klientów połączenia będą akceptowane przez jądro systemu.

- W pętli *while* umieszczam funkcję *accept()*, która powoduje, że proces serwera popada w stan uśpienia w oczekiwaniu na nadejście i zaakceptowanie połączenia z klientem.
- Po nawiązaniu połączenia używam funkcji *read()* i *write()* do komunikacji z danym klientem.

- **Opis powyższego programu (od strony klienta)**

- Za pomocą funkcji *socket()* tworzę lokalne gniazdo strumieniowe.

- W gniazdowej strukturze adresowej dla dziedziny UNIX'a (struktura *sockaddr_un* określona identyfikatorem *adres_serwera*) umieszczam adres gniazda, z którym proces klienta będzie chciał nawiązać połączenie. W tym przypadku jest to gniazdo serwera. Pole *sun_family* wypełniam wartością AF_UNIX, natomiast w polu *sun_path* wpisuję wartość *gniazdo_serwera*.

- Za pomocą funkcji *connect()* próbuję ustanowić połączeniem z gniazdem serwera wyznaczonym przez gniazdową strukturę adresową określoną identyfikatorem *adres_serwera*.

- Po nawiązaniu połączenia używam funkcji *read()* i *write()* do komunikacji z serwerem.

Drugi przykład jest analogiczny do pierwszego, z tą różnicą, że środowiskiem gniazd jest teraz Internet.

System komunikacyjny oparty na modelu klient-serwer.
Dziedzina : Internet
Typ gniazd : Strumieniowe

Pierwszy program : serwer
źródło : s2.c

```
#include <sys/types.h>
#include <sys/socket.h>
#include <stdio.h>
#include <sys/un.h>
#include <unistd.h>
#include <netinet/in.h>
#include <arpa/inet.h>

void blad()
{
  perror("Blad serwera.");
  exit(1);
}


int main()
{
  int gniazdo_serwera;
  int gniazdo_dla_klienta;
  int serwer_dlugosc;
  int klient_dlugosc;
  int test1,test2,test3,test4,test5;
  struct sockaddr_in adres_serwera;
  struct sockaddr_in adres_klienta;
  //Utworzenie gniazda serwera
  test1=gniazdo_serwera=socket(AF_INET, SOCK_STREAM,0);
  if (test1==-1) blad();

  //Zwiazanie gniazda serwera z adresem
  adres_serwera.sin_family=AF_INET;
  adres_serwera.sin_addr.s_addr=inet_addr("127.0.0.1");
  adres_serwera.sin_port=9734;
  serwer_dlugosc=sizeof(adres_serwera);
  test2=bind(gniazdo_serwera,(struct sockaddr *)&adres_serwera,serwer_dlugosc);
```

```
 if (test2==-1) blad();

 //Utworzenie kolejki dla oczekujacych polaczen
 test3=listen(gniazdo_serwera,5);
 if (test3==-1) blad();

 printf("Serwer nasluchuje...\n");

 //Komunikacja gniazda serwera z gniazdem klienta
 while(1)
   {
   char znak;

   //Przyj,ęcie polaczenia przez gniazdo serwera
   gniazdo_dla_klienta=accept(gniazdo_serwera,(struct
                              sockaddr *)&adres_klienta,&klient_dlugosc);

   test4=read(gniazdo_dla_klienta,&znak,1);
   if (test4>0) printf("Odebralem od klienta znak %c.\n",znak);
   else blad();

   znak='b';

   test5=write(gniazdo_dla_klienta,&znak,1);
   if (test5>0) printf("Wyslalem do klienta znak %c.\n",znak);
   else blad();

   close(gniazdo_dla_klienta);
   }
 close(gniazdo_serwera);
 exit(0);
}
```

Drugi program : klient
źródło : k2.c

```
#include <sys/types.h>
#include <sys/socket.h>
#include <stdio.h>
#include <sys/un.h>
#include <unistd.h>
#include <netinet/in.h>
#include <arpa/inet.h>

void blad()
{
  perror("Blad klienta.");
  exit(1);
}
```

```
int main()
{
  int gniazdo_klienta;
  int serwer_dlugosc;
  char znak='a';
  int test1,test2,test3,test4,test5;
  struct sockaddr_in adres_serwera;

  //Utworzenie gniazda klienta
  test1=gniazdo_klienta=socket(AF_INET, SOCK_STREAM,0);
  if (test1==-1) blad();


  //Ustalenie adresu serwera
  adres_serwera.sin_family=AF_INET;
  adres_serwera.sin_addr.s_addr=inet_addr("127.0.0.1");
  adres_serwera.sin_port=9734;
  serwer_dlugosc=sizeof(adres_serwera);

  //Polaczenie gniazda klienta z gniazdem serwera
  test1=connect(gniazdo_klienta,(struct sockaddr *)&adres_serwera,serwer_dlugosc);
  if (test1==-1) blad();

  //Komunikacja gniazda serwera z gniazdem klienta
  test4=write(gniazdo_klienta,&znak,1);
  if (test4>0) printf("Wyslalem do serwera znak %c.\n",znak);
  else blad();

  test5=read(gniazdo_klienta,&znak,1);
  if (test5>0) printf("Otrzymalem od serwera znak %c.\n",znak);
  else blad();

  close(gniazdo_klienta);
  exit(0);
}
```

W tym przykładzie serwer i klient pracują w sieci. Ale nie należy zakładać, że gniazda sieciowe bywają użyteczne tylko w sieci złożonej z wielu komputerów z zainstalowanym systemem UNIX (LINUX). Należy pamiętać, że nawet pojedynczy komputer połączonym z Internetem (za pomocą modemu) może korzystać z gniazd, aby porozumiewać z innymi komputerami. A co więcej można używać sieciowych programów na izolowanych komputerze, ponieważ większość maszyn z zainstalowanym system UNIX (LINUX) skonfigurowana jest do korzystania z sieci zwrotnej (ang. *loopback network)*, która zawiera tylko samą siebie. Sieć zwrotna składa się z jednego komputera, o nazwie *localhost* i adresie IP *127.0.0.1*. Ponieważ komputer na którym pisałem i testowałem ten program jest komputerem izolowanym (nie podłączonym do żadnej sieci), wykorzystałem w tym przykładzie adres sieci

zwrotnej.

- **Opis powyższego programu : od strony serwera.**

- Tworzę gniazdo domeny AF_INET za pomocą funkcji *socket().*

- Wypełniam internetową strukturę gniazdową w następujący sposób :

```
adres_serwera.sin_family=AF_INET;
adres_serwera.sin_addr.s_addr=inet_addr("127.0.0.1");
adres_serwera.sin_port=9734;
```

  Gniazdo będzie związane z wybranym przeze mnie portem (w tym przypadku jest to port o numerze 9734). Określony adres wskazuje, którym komputerom wolno się łączyć z gniazdem. Ponieważ wybrałem adres sieci zwrotnej, komunikacja będzie ograniczona do lokalnego komputera. Jeżeli chciałbym zezwolić na komunikację ze zdalnymi klientami, musiałbym skorzystać ze specjalnej wartości ***INADDR_ANY***, aby określić, że będę akceptował połączenia ze wszystkich interfejsów sieciowych, w które wyposażony jest mój komputer. Stała *INADDR_ANY* jest 32-bitową liczbą całkowitą i musi być umieszczona w polu *sin_addr.s_addr* struktury adresowej. Należy pamiętać, aby przetłumaczyć swoją wewnętrzną reprezentację liczb całkowitych na sieciowy porządek. Można wykorzystać omawianą już w trzecim rozdziale funkcję *htonl()*.

```
adres_serwera.sin_addr.s_addr=htonl(INADDR_ANY)
```

- Następnie wiążę gniazdo utworzone za pomocą funkcji *socket()* z adresem zdefiniowanym w powyższej strukturze. Służy do tego funkcja *bind().*

- Dalszy opis jest analogiczny do opisu serwera lokalnego z poprzedniego przykładu.

- **Opis powyższego programu : od strony klienta**

- Tworzę gniazdo domeny AF_INET za pomocą funkcji *socket().*

- W internetowej strukturze adresowej (struktura *sockaddr_in* określona identyfikatorem *adres_serwera*) podaję adres gniazda, z którym proces klienta będzie chciał nawiązać połączenie. Jest to serwer działający na hoście o adresie *127.0.0.1* i numerze portu *9734*. Wykorzystuje tutaj funkcję *inet_addr()*, aby zamienić tekstową reprezentację adresu IP na formę pozwalającą na adresowanie gniazd.

```
adres_serwera.sin_addr.s_addr=inet_addr("127.0.0.1");
```

- Dalszy opis jest analogiczny do opisu klienta lokalnego z poprzedniego przykładu.

Trzeci przykład opisuje dwa procesy komunikujące się ze sobą za pomocą gniazd datagramowych. Środowiskiem jest system LINUX

System komunikacyjny oparty na modelu klient-serwer.
Dziedzina : LINUX
Typ gniazd : Datagramowe

Pierwszy program : serwer
źródło : s3.c

```
#include <sys/types.h>
#include <sys/socket.h>
#include <stdio.h>
#include <sys/un.h>
#include <unistd.h>

void blad()
{
  perror("Blad serwera.");
  exit(1);
}

int main()
{
  int gniazdo_klienta;
  int gniazdo_serwera;
  int serwer_dlugosc;
  int klient_dlugosc;
  char znak;
  int test1,test2;
  struct sockaddr_un adres_serwera;
  struct sockaddr_un adres_klienta;

  //usuniecie starego gniazda serwera
  unlink("gniazdo_serwera");

  //Utworzenie gniazda serwera
  test1=gniazdo_serwera=socket(AF_UNIX, SOCK_DGRAM,0);
  if (test1==-1) blad();

  //Zwiazanie gniazda serwera z adresem
  adres_serwera.sun_family=AF_UNIX;
  strcpy(adres_serwera.sun_path,"gniazdo_serwera");
  serwer_dlugosc=sizeof(adres_serwera);
  test2=bind(gniazdo_serwera,(struct sockaddr *)&adres_serwera,serwer_dlugosc);
  if (test2==-1) blad();
```

```
  //Serwer czeka na wiadomosc od klienta.
  recvfrom(gniazdo_serwera,&znak,1,0,(struct sockaddr *)&adres_klienta,&klient_dlugosc);
  printf("Otrzymalem od klienta znak %c.\n",znak);

  znak='b';

  //Serwer wysyla klientowi zmodyfikowana wiadomosc
  sendto(gniazdo_serwera,&znak,1,0,(struct sockaddr *)&adres_klienta,klient_dlugosc);
  printf("Wyslalem do klienta znak %c.\n",znak);

  close(gniazdo_serwera);
  exit(0);
}
```

Drugi program : klient
źródło : k3.c

```
#include <sys/types.h>
#include <sys/socket.h>
#include <stdio.h>
#include <sys/un.h>
#include <unistd.h>

void blad()
{
  perror("Blad klienta.");
  exit(1);
}

int main()
{
  int gniazdo_klienta;
  int serwer_dlugosc;
  int klient_dlugosc;
  char znak='a';
  int test1,test2;
  struct sockaddr_un adres_serwera;
  struct sockaddr_un adres_klienta;

  //Usuniecie starego gniazda klienta
  unlink("gniazdo_klienta");

  //Utworzenie gniazda klienta
  test1=gniazdo_klienta=socket(AF_UNIX, SOCK_DGRAM,0);
  if (test1==-1) blad();

  //Zwiazanie gniazda klienta z adresem
  adres_klienta.sun_family=AF_UNIX;
  strcpy(adres_klienta.sun_path,"gniazdo_klienta");
  klient_dlugosc=sizeof(adres_klienta);
```

```
 test2=bind(gniazdo_klienta,(struct sockaddr *)&adres_klienta,klient_dlugosc);
 if (test2==-1) blad();

 //Ustalenie adresu serwera
 adres_serwera.sun_family=AF_UNIX;
 strcpy(adres_serwera.sun_path,"gniazdo_serwera");
 serwer_dlugosc=sizeof(adres_serwera);

 //Klient wysyla wiadomosc do serwera
 sendto(gniazdo_klienta,&znak,1,0,(struct sockaddr *)&adres_serwera,serwer_dlugosc);
 printf("Wyslalem do serwera znak %c.\n",znak);

 //Klient odbiera od serwera zmodyfikowana wiadomosc
 recvfrom(gniazdo_klienta,&znak,1,0,(struct sockaddr *)&adres_serwera,&serwer_dlugosc);
 printf("Otrzymalem od serwera znak %c.\n",znak);

 close(gniazdo_klienta);
 exit(0);
}
```

- **Opis powyższego programu : od strony serwera.**

- Tworzone jest gniazdo datagramowe za pomocą funkcji *socket().*
- Tworzony jest adres serwera (plik o nazwie *adres_serwera)* i wiązany z gniazdem za pomocą wywołania *bind()*.
- Proces serwera oczekuje na datagram przywołując funkcję *recvfrom()*.
- Proces serwera przetwarza otrzymany datagram.
- Wynik jest przesyłany z powrotem do klienta za pomocą funkcji *sendto()*, używając adresu klienta (nadawcy), który serwer otrzymał po wykonaniu funkcji *recvfrom()*.

- **Opis powyższego programu : od strony klienta**

- Tworzone jest gniazdo datagramowe za pomocą funkcji *socket().*
- Tworzony jest adres klienta (plik o nazwie *: adres_klienta)* i wiązany z gniazdem za pomocą wywołania *bind().*
- Proces klienta wysyła datagram do serwera poprzez wywołanie funkcji *sendto().*
- Proces klienta oczekuje na odpowiedź od serwera używając funkcji *recvfrom()*.

Ostatni przykład opisuje dwa procesy komunikujące się ze sobą za pomocą gniazd datagramowych. Teraz środowiskiem jest Internet.

System komunikacyjny oparty na modelu klient-serwer.
Dziedzina : Internet
Typ gniazd : Datagramowe

Pierwszy program : serwer
źródło : s4.c

```
#include <sys/types.h>
#include <sys/socket.h>
#include <stdio.h>
#include <sys/un.h>
#include <unistd.h>
 #include <netinet/in.h>
#include <arpa/inet.h>

void blad()
{
  perror("Blad serwera.");
  exit(1);
}

int main()
{
  int gniazdo_klienta;
  int gniazdo_serwera;
  int serwer_dlugosc;
  int klient_dlugosc;
  char znak;
  int test1,test2;
  struct sockaddr_in adres_serwera;
  struct sockaddr_in adres_klienta;


  //Utworzenie gniazda serwera
  test1=gniazdo_serwera=socket(AF_INET, SOCK_DGRAM,0);
  if (test1==-1) blad();

  //Zwiazanie gniazda serwera z adrese
  adres_serwera.sin_family=AF_INET;
  adres_serwera.sin_addr.s_addr=inet_addr("127.0.0.1");
  adres_serwera.sin_port=9734;
  serwer_dlugosc=sizeof(adres_serwera);
  test2=bind(gniazdo_serwera,(struct sockaddr *)&adres_serwera,serwer_dlugosc);
  if (test2==-1) blad();
```

```
  //Serwer czeka na wiadomosc od klienta.
  recvfrom(gniazdo_serwera,&znak,1,0,(struct sockaddr *)&adres_klienta,&klient_dlugosc);
  printf("Otrzymalem od klienta znak %c.\n",znak);
  znak='b';

  //Serwer wysyla klientowi zmodyfikowana wiadomosc
  sendto(gniazdo_serwera,&znak,1,0,(struct sockaddr *)&adres_klienta,klient_dlugosc);
  printf("Wyslalem do klienta znak %c.\n",znak);

  close(gniazdo_serwera);
  exit(0);
}
```

Drugi program : klient
źródło : k4.c

```
#include <sys/types.h>
#include <sys/socket.h>
#include <stdio.h>
#include <sys/un.h>
#include <unistd.h>
#include <netinet/in.h>
#include <arpa/inet.h>

void blad()
{
  perror("Blad serwera.");
  exit(1);
}

int main()
{
  int gniazdo_klienta;
  int gniazdo_serwera;
  int serwer_dlugosc;
  int klient_dlugosc;
  char znak='a';
  int test1,test2;
  struct sockaddr_in adres_serwera;
  struct sockaddr_in adres_klienta;


  //Utworzenie gniazda klienta
  test1=gniazdo_klienta=socket(AF_INET, SOCK_DGRAM,0);
  if (test1==-1) blad();
  //Zwiazanie gniazda klienta z adresem
  adres_klienta.sin_family=AF_INET;
  adres_klienta.sin_addr.s_addr=inet_addr("127.0.0.1");
  adres_klienta.sin_port=9735;
  klient_dlugosc=sizeof(adres_serwera);
```

```
 test2=bind(gniazdo_klienta,(struct sockaddr *)&adres_klienta,klient_dlugosc);
 if (test2==-1) blad();

 //Ustalenie adresu serwera
 adres_serwera.sin_family=AF_INET;
 adres_serwera.sin_addr.s_addr=inet_addr("127.0.0.1");
 adres_serwera.sin_port=9734;
 serwer_dlugosc=sizeof(adres_serwera);

  //Serwer wysyla klientowi zmodyfikowana wiadomosc
 sendto(gniazdo_klienta,&znak,1,0,(struct sockaddr *)&adres_serwera,serwer_dlugosc);
 printf("Wyslalem do serwera znak %c.\n",znak);

 //Serwer czeka na wiadomosc od klienta.
 recvfrom(gniazdo_klienta,&znak,1,0,(struct sockaddr *)&adres_serwera,&serwer_dlugosc);
 printf("Otrzymalem od serwera znak %c.\n",znak);

 close(gniazdo_klienta);
 exit(0);
}
```

- **Opis powyższego programu : od strony serwera.**

- Tworzone gniazdo datagramowe za pomocą funkcji *socket().*
- Tworzony jest adres serwera (adres IP : *127.0.0.1* oraz numer portu : *9734)* i wiązany z gniazdem za pomocą wywołania *bind().*
- Proces serwera oczekuje na datagram przywołując funkcję *recvfrom().*
- Proces serwera przetwarza otrzymany datagram.
- Wynik jest przesyłany z powrotem do klienta za pomocą funkcji *sendto()*, używając adresu klienta (nadawcy), który serwer otrzymał po wykonaniu funkcji *recvfrom()*.

- **Opis powyższego programu : od strony klienta.**

- Tworzone jest gniazdo datagramowe za pomocą funkcji *socket().*
- Tworzony jest adres klienta (adres IP : *127.0.0.1* oraz numer portu : *9735)* i wiązany z gniazdem za pomocą wywołania *bind().*
- Proces klienta wysyła datagram do serwera poprzez wywołanie funkcji systemowej *sendto()* i oczekuje na odpowiedź od serwera, używając funkcji *recvfrom()*.

Rozdział 5. Więcej o gniazdach

5.1. Opcje gniazd

Do ustawiania opcji gniazda i ich odczytywania są używane dwie funkcje. Program wywołuje funkcję *setsockopt()*, aby wpisać wartość opcji do gniazda, a funkcję *getsockopt()*, aby uzyskać aktualne wartości opcji. Opcje są głównie wykorzystywane do obsługi szczególnych sytuacji związanych z gniazdami (na przykład, aby kontrolować sposób zamykania gniazda po przywołaniu funkcji *close())*.

- **Pobranie informacji o ustawieniach opcji gniazd**

Możliwość zdobycia informacji o wartościach przypisanych opcjom danego gniazd jest dla programisty niezwykle ważne. Funkcją, która pozwala na badanie ustawień opcji gniazda, jest *getsockopt().*

```
#include <sys/types.h>
#include <sys/socket.h>

int getsockopt(int socket, int level, int optname, void *optval, socklen_t *optlen);
```

Argument *socket* oznacza deskryptor gniazda, które ma być sprawdzone. Paremtr *level* określa poziom protokołu, dla którego mają być sprawdzone opcje. Najczęściej używa się jednego z dwóch poziomów :

- SOL_SOCKET, aby sięgnąć do opcji z poziomu gniazda.
- SOL_TCP, aby sięgnąć do opcji z poziomu protokołu TCP.

Do argumenty *optname* należy wstawić nazwę opcji, która ma być sprawdzona. W tabeli 5.1 zamieściłem niektóre z możliwych kombinacji opcji i poziomów protokołu. Wartość przypisaną danej opcji obierzemy dzięki parametrowi **optval*, który jest wskaźnikiem do bufora przechowującego informację o tej opcji. Wskaźnik *optlen* oznacza zarówno długość bufora odbierającego dane, jak i długość zwracanych wartości opcji.

Funkcja zwraca wartość zero, gdy uda się wykonać zadanie. W przypadku błędu funkcja zwróci wartość -1, a przyczynę błędu można będzie ustalić, badając zmienną *errno*.

Poziom protokołu	*Nazwa opcji*
SOL_SOCKET	SO_REUSEADDR
SOL_SOCKET	SO_KEEPALIVE
SOL_SOCKET	SO_LINGER
SOL_SOCKET	SO_DEBUG
SOL_SOCKET	SO_OOBINLINE
SOL_SOCKET	SO_SNDBUF
SOL_SOCKET	SO_RCVBUF
SOL_SOCKET	SO_TYPE
SOL_SOCKET	SO_ERRROR
SOL_TCP	SO_NODELAY

Tabela 5.1. Poziomy protokołu i nazwy opcji.

- **Definiowanie opcji gniazd**

Opcje gniazd ustawiamy używając funkcji *setsockopt()*.

```
#include <sys/socket.h>
#include <sys/types.h>

int setsockopt(int socket, int level, int optname, const void *optval, socklen_t optlen);
```

Pierwszy argument musi odnosić się do otwartego deskryptora gniazda. Argument *level* definiuje poziom gniazda zmienianej opcji. Parametr *optname* określa nazwę opcji, którą chcemy ustawić. Argument *optval* jest wskaźnikiem do wartości, która stanie się nową wartością dla danej opcji. Ostatni parametr definiuje długość opcji w bajtach. Funkcja zwraca wartość 0 w razie pomyślnego wykonania lub -1 w przeciwnym przypadku.

- **Opis wybranych opcji gniazd**

- ***SO_TYPE*** - *t*a opcja nie może być zmieniona. Jedyne co można zrobić, to sprawdzić jej wartość, która określa typ gniazda o danym deskryptorze.

- ***SO_LINGER*** – jej zadaniem jest kontrola tego, w jaki sposób gniazdo jest zamykane po przywołaniu funkcji *close()*. Tryby tej opcji są kontrolowane przez strukturę *linger.* Zmienna *l_onoff* pełni funkcję wartości logicznej, dla której wartość różna od zera oznacza włączenie opcji, a wartość zerowa jej wyłączenie. Istnieją trzy możliwe ustawienia dla opcji SO_LINGER :

```
struct linger
{
  int l_onoff  // 0 – opcja wyłączona, niezero – opcja włączona
  int l_linger;// czas zwlekania (w sekundach)
}
```

- Przypisanie zmiennej *l_onoff* wartości 0 powoduje, że zmienna *l_linger* jest ignorowana i działa domyślny sposób zamykania gniazda przy pomocy funkcji *close()*.

- Przypisanie zmiennej *l_onoff* wartości niezerowej sprawi, że zmienna *l_linger* będzie miała wpływ na sposób zamykania gniazda. Jeżeli wartość zmiennej *l_linger* ustawimy na różną od zera, to wartość jej przypisana podaje czas (w sekundach), przez który funkcja *close()* będzie się starać wysłać oczekujące dane i zamknąć gniazdo.

- Ustawienie zmiennej *l_onoff* na wartość niezerową i przypisanie zmiennej *l_linger* wartości zero spowoduje, że funkcja *close()* natychmiast zerwie połączenie i pozbędzie się wszelkich danych oczekujących na wysłanie.

- ***SO_DEBUG*** - zezwala na korzystanie z diagnostyki prowadzonej na niskim poziomie w jądrze lub tego zabrania. W przypadku zezwolenia jądro systemu prowadzi historię ostatnio wysyłanych lub odebranych pakietów przez gniazdo o danym deskryptorze.

5.2. Serwery współbieżne

W poprzednich rozdziałach zajmowałem się implementacją systemów opartych na architekturze “klient-serwer”, w których serwer był iteracyjny, czyli mógł aktualnie obsługiwać tylko jednego klienta. Teraz omówię serwery współbieżne, czyli takie, które w danej chwili potrafią obsłużyć jednocześnie wielu klientów. Najpierw należy zdefiniować funkcję **fork()**. Wywołanie tej funkcji jest jedynym sposobem utworzenia nowego procesu. Funkcja ta tworzy kopię tego procesu, który wywołał funkcję *fork()*. Proces wywołujący tę funkcję nazywa się **procesem macierzystym** (ang. *parent process*), a nowy proces utworzony przez funkcję *fork()* nazywany jest **procesem potomnym** (ang. *child process*). Specyfika tej funkcji polega na tym, że związane są z nią dwa powroty. Najpierw powraca ona do procesu macierzystego, który ją wywołał i przekazuje mu identyfikator nowo utworzonego procesu potomnego. Następnie powraca do procesu potomnego i przekazuje mu wartość zero. Dlatego na podstawie wartości przekazanych przez funkcję *fork()* można odróżnić proces macierzysty od potomnego. W praktyce proces macierzysty wywołuje funkcję *fork()*, aby on mógł się zająć jedną operacją, podczas gdy proces potomny (jego kopia) mógłby wykonywać inne zadanie. Ważny jest tutaj fakt, że

wszystkie otwarte deskryptory (między innymi deskryptory gniazd), które były otwarte w procesie macierzystym przed wywołaniem funkcji *fork()*, będą po powrocie z niej dostępne również dla procesu potomnego. Właśnie ze względu na powyższą własność tej funkcji, zdecydowałem się wykorzystać ją do implementacji serwera współbieżnego.

Przykład : System komunikacyjny oparty na modelu klient-serwer z obsługą wielu klientów jednocześnie.
Dziedzina : UNIX
Typ gniazd : Strumieniowe

Pierwszy program : serwer
źródło : s5.c

```
#include <sys/types.h>
#include <sys/socket.h>
#include <stdio.h>
#include <sys/un.h>
#include <unistd.h>
#include <signal.h>

void blad()
{
  perror("Blad serwera.");
  exit(1);
}
int main()
{
  int gniazdo_serwera;
  int gniazdo_dla_klienta;
  int serwer_dlugosc;
  int klient_dlugosc;
  int test1,test2,test3,test4,test5;
  char znak;
  struct sockaddr_un adres_serwera;
  struct sockaddr_un adres_klienta;

  //Usuniecie starego gniazda
  unlink("gniazdo_serwera");

  //Utworzenie gniazda serwera
  test1=gniazdo_serwera=socket(AF_UNIX, SOCK_STREAM,0);
  if (test1==-1) blad();

  //Zwiazanie gniazda serwera z adresem
  adres_serwera.sun_family=AF_UNIX;
  strcpy(adres_serwera.sun_path,"gniazdo_serwera");
  serwer_dlugosc=sizeof(adres_serwera);
```

```
  test2=bind(gniazdo_serwera,(struct sockaddr *)&adres_serwera,serwer_dlugosc);
  if (test2==-1) blad();

  //Utworzenie kolejki dla oczekujacych polaczen
  test3=listen(gniazdo_serwera,5);
  if (test3==-1) blad();
  signal(SIGCHLD,SIG_IGN);

  printf("Serwer nasluchuje...\n");

  //Komunikacja gniazda serwera z gniazdem klienta
  while(1)
    {
    //Przyjęcie polaczenia przez gniazdo serwera
gniazdo_dla_klienta=accept(gniazdo_serwera,(struct
                          sockaddr *)&adres_klienta,&klient_dlugosc);
    if (fork()==0)
      {
      printf("Obsluguje nowego klienta.\n");
      test4=read(gniazdo_dla_klienta,&znak,1);
      if (test4>0) printf("Odebralem do klienta znak %c.\n",znak);
      else blad();

      sleep(5);
      znak='b';

      test5=write(gniazdo_dla_klienta,&znak,1);
      if (test5>0) printf("Wysylam dane do klienta znak %c.\n",znak);
      else blad();
      close(gniazdo_dla_klienta);
      exit(0);
      }
    else
      {
        close(gniazdo_dla_klienta);
      }
    }
  close(gniazdo_serwera);
  exit(0);
}
```

Drugi program : klient
źródło : k5.c
Program wygląda analogicznie jak k.c

Po ustanowieniu połączenia i powrocie z funkcji *accept()* serwer wywołuje funkcję *fork()*, a następnie proces potomny obsługuje klienta przez gniazdo połączone, podczas gdy proces macierzysty czeka na drugie

połączenie przez gniazdo nasłuchujące. Proces macierzysty zamyka gniazdo połączone, ponieważ aktualnego klienta obsługuje teraz proces potomny.

5.3. Nienazwane gniazda w dziedzinie UNIX'a

Gniazdo nie zawsze musi mieć adres. Na przykład funkcja *socketpair()* tworzy dwa gniazda połączone ze sobą, jednak nie mające adresów. Tego typu gniazda stosuje się tylko wewnątrz tego samego systemu operacyjnego.

```
#include <sys/types.h>
#include <sys/socket.h>

int socketpair(int family, int type, int protocol, int sockfd[2]);
```

Funkcja *socketpair()* tworzy dwa gniazda, które są następnie łączone ze sobą. Argument *family* musi mieć wartość AF_UNIX (AF_LOCAL), argument *protocol* musi mieć wartość 0. Natomiast wartością argumentu *type* może być albo SOCK_STREAM albo SOCK_DGRAM. Funkcja *socketpair()* tworzy dwa deskryptory gniazd przekazywane jako *sockfd[0]* oraz *sockfd[1]*. Dwa utworzone gniazda nie mają nazwy – nie wiążemy ich z żadnym adresem. Należy pamiętać, że funkcja *socketpair()* działa tylko w dziedzinie UNIX'a, zatem nie możemy jej uruchamiać z parametrem *family* mającym wartość AF_INET. Funkcję tę ilustruje poniższy przykład :

Przykład : utworzenie pary nienazwanych gniazd w dziedzinie Unix'a i komunikacja między nimi.
Dziedzina : LINUX
Typ gniazd : Nienazwane
źródło : sp.c

```
#include <sys/types.h>
#include <sys/socket.h>

void blad()
{
  perror("Blad programu");
  exit(1);
}

main()
{
   int gniazdo[2];
   char znak;
   int test1;
   int test2;
```

```
    znak='a';

  test1=socketpair(AF_UNIX, SOCK_STREAM, 0, gniazdo);
  if (test1==-1) blad();

  test2=fork();
  if (test2==0)
    {
     close(gniazdo[1]);
     read(gniazdo[0],&znak,1);
     printf("Uzytkownik 2 : Odebralem wiadomosc : %c.\n",znak);
     close(gniazdo[0]);
     exit(0);
    }
  else if (test2>0)
    {
     close(gniazdo[0]);
     printf("Uzytkownik 1 : Wysylam wiadomosc : %c.\n",znak);
     write(gniazdo[1],&znak,1);
     close(gniazdo[1]);
     wait();
     exit(0);
    }
  else blad();
}
```

- **Opis działania programu**

- Proces macierzysty tworzy za pomocą funkcji *fork()* proces potomny.
- Proces macierzysty wysyła do procesu potomnego wiadomość 'a', używając gniazda *gniazdo[1]*.
- Proces potomny odbiera wysłaną wiadomość, używając gniazda *gniazdo[0]*.

Dodatek A.
Błędy związane z gniazdami

- **ENOTSTOCK**

Na deskryptorze pliku, który nie jest gniazdem, chciano wykonać operację charakterystyczną dla gniazd.

- **EDESTADDRREQ**

Próba wysłania danych przez gniazdo bez podania adresu docelowego. Ten błąd występuje tylko w gniazdach datagramowych.

- **EPROTOTYPE**

Podano niewłaściwy protokół.

- **ENOPROTOOPT**

Próba ustawienia niepoprawnych opcji.

- **EPROTONOSUPPORT**

Żądanie dotyczące niewspieranego protokołu.

- **ESOCKTNOSUPPORT**

Próba stworzenia niewspieranego typu gniazda.

- **EPFNOSUPPORT**

Podana niewspierana rodzina protokołów.

- **EAFNOSUPPORT**

Podana niewspierana rodzina adresów.

- **EADDRINUSE**

Żądany adres jest już używany i nie może być przydzielony.

- **EADDRNOTAVAIL**

Żądanie dotyczące niedostępnego adresu.

- **ENETDOWN**

Zerwane połączenie sieciowe.

- **ENETUNREACH**

Niedostępna sieć.

- **ENETRESET**

Połączenie zerwane z powodu zmiany konfiguracji sieci.

- **ECONNABORTED**

Połączenie zerwane przez program.

- **ECONNRESET**

Zerwanie połączenia przez drugą stronę. Zazwyczaj wtedy, gdy komputer jest ponownie włączony.

- **ENOBUFS**

Zbyt mało dostępnego miejsca w buforach na obsłużenie połączenia.

- **EISCONN**

Z tym gniazdem jest już nawiązane połączenie.

- **ENOTCONN**

Zanim będzie wykonana operacja, musi być nawiązane połączenie.

- **ETIMEDOUT**

Minął czas na nawiązanie połączenia.

- **ECONNREFUSED**

Stacja zdalna odmówiła połączenia.

- **EHOSTDOWN**

Stacja zdalna nie jest w sieci.

- **EHOSTUNREAD**

Nieosiągalna stacja zdalna

Dodatek B. Zawartość dyskietki

- k.c prosty klient lokalny, transmisja połączeniowa
- s.c prosty serwer lokalny, transmisja połączeniowa
- k2.c prosty klient sieciowy, transmisja połączeniowa (TCP)
- s2.c prosty serwer sieciowy, transmisja połączeniowa (TCP)
- k3.c prosty klient lokalny (transmisja bezpołączeniowa)
- s3.c Prosty serwer lokalny (transmisja bezpołączeniowa)
- k4.c Prosty klient sieciowy (transmisja UDP)
- s4.c Prosty serwer sieciowy (transmisja UDP)
- k5.c Klient lokalny, transmisja połączeniowa
- s5.c serwer współbieżny, transmisja połączeniowa
- bind.c opis funkcji *bind() [dziedzina UNIX]*
- bind2.c opis funkcji *bind() [dziedzina Internet]*
- ghs.c opis funkcji gethostbyname()
- sp.c opis funkcji socketpair()

Bibliografia

[1] Douglas E. Comer: *Sieci komputerowe i intersieci*, Wydawnictwa Naukowo-Techniczne, Warszawa 2000, 2001;

[2] Warren W. Gay: *Linux - gniazda w programowaniu w przykładach*, Wydawnictwo “MIKOM”, Warszawa 2001;

[3] Craig Hunt: *TCP/IP – administracja sieci*;

[4] Michael K. Johnson, Erik W. Troan: *Oprogramowanie użytkowe w systemie Linux*;

[5] Neil Matthew, Richard Stones: *Linux – programowanie*, Wydawnictwo RM, Warszawa 1999;

[6] Mark Mitchell i in.: *Programowanie dla zaawansowanych*;

[7] Jerzy Skurczyński: *Programowanie współbieżne*, Instytut Matematyki Uniwersytetu Gdańskiego, Gdańsk 2000;

[8] W. Richard Stevens: Programowanie zastosowań sieciowych w systemie Unix, Wydawnictwa Naukowo-Techniczne, Warszawa 1995, 1996, 1998;

[9] W. Richard Stevens: *Unix – programowanie usług sieciowych*, Wydawnictwa Naukowo-Techniczne, Warszawa 2000;