Perl. Tworzenie aplikacji sieciowych

Autor: Lincoln D. Stein
Tłumaczenie: Robert Gębarowski
ISBN: 83-7197-604-6
Tytuł oryginału: Network Programming with Perl
Format: B5, stron: 834
Przykłady na ftp: 76 kB

Programowanie aplikacji sieciowych to jedna z tych dziedzin, z którą język Perl radzi sobie doskonale. Zwłaszcza, gdy czas nagli, a potrzebujemy napisać program spełniający funkcje serwera czy też klienta sieciowego, docenimy zalety Perla: zwięzłość kodu, dostęp do wielu wbudowanych procedur i setek modułów rozszerzających ten język oraz szybkość z jaką w Perlu tworzy się gotowe, działające aplikacje.

Książka poświęcona jest głównie protokołowi TCP/IP, będącemu fundamentem funkcjonowania Internetu. Omówiono w niej:

- protokół TCP oraz interfejs programowania modułu IO::Socket,
- protokół SMTP i wysyłanie poczty elektronicznej z załącznikami multimedialnymi,
- protokoły POP, IMAP i NNTP do odbioru i przetwarzania poczty elektronicznej,
- protokół FTP, protokół HTTP i moduł LWP do komunikacji z serwerami WWW,
- serwery rozwidlające się oraz demony inetd systemów UNIX i Windows,
- programowanie wielowątkowe w Perlu,
- protokół UDP i serwery oparte na tym protokole,
- komunikację między procesami za pośrednictwem gniazd domeny UNIX.

Autor książki, Lincoln Stein, to prawdziwy guru programowania sieciowego w Perlu. Wystarczy tylko wspomnieć, iż jest on autorem modułu CGI.pm, powszechnie używanego przy pisaniu skryptów CGI, a także autorem licznych książek na temat tego języka.

Wydawnictwo Helion
ul. Chopina 6
44-100 Gliwice
tel. (32)230-98-63
e-mail: helion@helion.pl

Spis treści

Rozdział 4.

Protokół TCP

W tym rozdziale przyjrzymy się niezwykle solidnemu, zorientowanemu na połączenie protokołowi sterowania transmisją strumienia bajtów TCP (ang. *Transmission Control Protocol*). Jego właściwości sprawiają, że praca z gniazdami TCP przypomina pracę z uchwytami plików i potokami. Otwarcie gniazda TCP umożliwia przesłanie przez nie danych za pomocą funkcji `print()` lub `syswrite()` albo odczytanie danych przy użyciu operatora <> czy funkcji `read()` lub `sysread()`.

Klient usługi echo w protokole TCP

Zaczniemy od opracowania niewielkiego klienta TCP; będzie to zadanie znacznie bardziej skomplikowane od wszystkich zaprezentowanych w dotychczas omówionych przykładach. Na ogół klient jest odpowiedzialny za aktywne zainicjowanie swego połączenia ze zdalna usługą. Zarys tego procesu został już naszkicowany w rozdziale 3. I tak, gwoli przypomnienia, klient TCP musi podjąć następujące kroki:

1. **Wywołaj funkcję `socket()`, by utworzyć gniazdo**. Z pomocą tej funkcji klient tworzy gniazdo typu strumieniowego (ang. *a stream-type socket*) w domenie `INET` (Internet), używające protokółu TCP.
2. **Wywołaj funkcję `connect()`, by połączenie:z równorzędnym zdalnym partnerem**. Klient określa pożądany adres docelowy i łączy z nim gniazdo za pomocą funkcji `connect()`.
3. **Wykonaj operacje wejścia-wyjścia na gnieździe**. Klient wywołuje rozmaite operacje wejścia i wyjścia, by komunikować się poprzez gniazdo.
4. **Zamknij gniazdo**. Po zakończeniu wszystkich operacji wejścia-wyjścia, klient może zamknąć gniazdo, używając funkcji `close()`.

Przedstawiony w tym podrozdziale przykład aplikacji to prosty klient usługi **echo** TCP. Usługa echo wykonywana standardowo na wielu hostach pracujących w systemie UNIX nie jest skomplikowana. Oczekuje na nadchodzące połączenie, przyjmuje je, a następnie powtarza każdy otrzymany bajt. Trwa to do momentu zamknięcia połączenia przez klienta.

Wypróbowanie przykładowego skryptu będzie wymagało skorzystania z serwera usługi echo. Można użyć na przykład serwera *malenstwo.iinf.polsl.gliwice.pl*.

Wydruk 4.1 pokazuje zawartość skryptu *tcp_echo_cli1.pl*. Najpierw przyjrzyjmy się samemu kodowi skryptu.

Wydruk 4.1. *Klient usługi echo w protokole TCP*

```
0   #!/usr/bin/perl
1   # plik: tcp_echo_cli1.pl
2   # uzycie: tcp_echo_cli1.pl [host] [port]
3   # Klient uslugi echo, wersja TCP

4   use strict;
5   use Socket qw(:DEFAULT :crlf);
6   use IO::Handle;
7   my ($bytes_out,$bytes_in) = (0,0);

8   my $host = shift || 'localhost';
9   my $port = shift || getservbyname('echo','tcp');

10   my $protocol = getprotobyname('tcp');
11   $host = inet_aton($host) or die "$host: host nieznany";

12   socket(SOCK, AF_INET, SOCK_STREAM, $protocol) or die "fiasko socket(): $!";
13   my $dest_addr = sockaddr_in($port,$host);
14   connect(SOCK,$dest_addr) or die "fiasko connect(): $!";

15   SOCK->autoflush(1);

16   while (my $msg_out = <>) {
17     chomp $msg_out;
18     $msg_out .= CRLF;
19     print SOCK $msg_out;
20     my $msg_in = <SOCK>;
21     print $msg_in;

22     $bytes_out += length($msg_out);
23     $bytes_in  += length($msg_in);
24   }

25   close SOCK;
26   print STDERR "bajty_wysłane = $bytes_out, bajty_odebrane = $bytes_in\n";
```

Wiersze 1 – 6: Ładuj moduły. Na tym etapie włączamy opcję ścisłego sprawdzania składni i ładujemy moduły `Socket` oraz `IO::Handle`. Modułu `Socket` używamy dla stałych związanych z gniazdem, zaś modułu `IO::Handle` z uwagi na dodawaną przez niego metodę `autoflush()`.

Wiersz 7: Zadeklaruj zmienne globalne. Teraz tworzymy dwie zmienne globalne do przechowywania rejestru liczby wysłanych i otrzymanych bajtów.

Wiersze 8 – 9: Przetwarzaj argumenty wiersza poleceń. Z wiersza polecenia odczytujemy nazwę docelowego hosta i numer docelowego portu. Jeśli host nie jest określony, przyjmujemy domyślnie nazwę lokalnego hosta, `localhost`. Jeśli nie podano

numeru portu, wykorzystujemy funkcję `getservbyname()`, by sprawdzić numer portu dla usługi echo.

Wiersze 10 – 11: Odszukaj numer protokołu i utwórz upakowany adres IP. Na tym etapie wykorzystujemy funkcję `getprotobyname()` dla uzyskania numeru protokołu TCP, którego użyje funkcja `socket()`. Następnie stosujemy `inet_aton()` do zamiany nazwy hosta do postaci upakowanego adresu IP, który można wykorzystać z funkcją `sockaddr_in()`.

Wiersz 12: Utwórz gniazdo**. Wywołujemy teraz funkcję** `socket()`, by utworzyć uchwyt pliku gniazda o nazwie `SOCK` — podobnie jak w przykładzie z rozdziału 3. (wydruk 3.1). Przekazujemy argumenty określając rodzinę adresów internetowych `AF_INET`, strumieniowy typ gniazda strumieniowego `SOCK_STREAM` i odszukany wcześniej numer protokołu TCP.

Wiersz 13 – 14: Utwórz adres docelowy i połącz z nim gniazdo. Teraz wykorzystujemy funkcję `sockaddr_in()` do utworzenia upakowanego adresu, zawierającego docelowy adres IP i numer portu. Jest on teraz adresem docelowym dla wywołania `connect()`. W razie powodzenia `connect()` zwraca wartość logiczną *prawda*. W przeciwnym przypadku operacja zostaje zakończona wyświetleniem komunikatu o błędzie.

Wiersz 15: Włącz dla gniazda tryb automatycznego opróżniania. Chcemy, aby dane zapisane do gniazda zamiast zajmować miejsce w lokalnym buforze były z niego natychmiast usuwane. Wywołujemy zatem metodę gniazda `autoflush()`, by włączyć tryb automatycznego opróżniania. Metoda automatycznego opróżniania jest dostępną dzięki modułowi `IO::Handle`.

Wiersze 16 – 24: Główna pętla. Teraz rozpoczynamy małą pętlę. Przy każdym jej wykonaniu odczytujemy wiersz tekstu ze standardowego wejścia, a następnie wysyłamy go do gniazda (`SOCK`), przesyłając ów wiersz do zdalnego hosta. Następnie, używając operatora <>, odczytujemy wiersz odpowiedzi z serwera i drukujemy go na standardowe wyjście. Przy każdorazowym przejściu przez pętlę zliczamy liczbę wysłanych i otrzymanych bajtów, aż do osiągnięcia znaku końca pliku (`EOF`) na standardowym wejściu.

Wiersze 25 – 26: Zamknij gniazdo i podaj statystykę. Po zakończeniu pętli zamykamy gniazdo i drukujemy na standardowym urządzeniu wyjścia błędu naszą statystykę wysłanych i przyjętych bajtów.

Sesja z użyciem skryptu *tcp_echo_cli1.pl* wygląda mniej więcej tak:

```
% tcp_echo_cli1.pl
Czy masz się dobrze Panie Bobrze?
Czy masz się dobrze Panie Bobrze?
Tutaj słychać echo.
Tutaj słychać echo.
Jo-di-lej-io-ou!
Jo-di-lej-io-ou!
^D
bajty_wysłane = 61, bajty_odebrane = 61
```

Symbol ^D w przedostatnim wierszu zapisu wskazuje na punkt, w którym znudziła mi się zabawa i nacisnąłem kombinację klawiszy powodującą zakończenie wprowadzania danych. W systemach Windows będzie to kombinacja *Ctrl+Z* (^Z).

Funkcje gniazda związane z wychodzącymi połączeniami

Teraz przyjrzymy się dokładniej funkcjom związanym z tworzeniem gniazd i ustanawianiem wychodzących połączeń TCP.

$boolean = socket(SOCKET,$domain,$type,$protocol)

Dla danej nazwy uchwytu pliku (SOCKET), domeny ($domain), typu ($type) i protokołu ($protocol) metoda socket() tworzy nowe gniazdo oraz kojarzy je z podaną nazwą uchwytu pliku. W razie powodzenia funkcja zwraca wartość logiczną *prawda*. W przypadku niepowodzenia socket() zwraca zapis undef i pozostawia komunikat o błędzie w $!. Domena, typ oraz protokół są małymi liczbami całkowitymi. Odpowiednimi wartościami dla domeny i typu są stałe zdefiniowane w module Socket, natomiast wartość protokołu musi być określona poprzez wywołanie getprotobyname() w czasie wykonywania. Typowy wzorzec do tworzenia gniazd TCP wygląda tak:

```
socket(SOCK,AF_INET,SOCK_STREAM, scalar getprotobyname('tcp'))
```

Funkcja getprotobyname() została tu umieszczona w kontekście skalarnym, tak, by zwróciła pojedynczy wynik określający numer protokołu.

$boolean = connect(SOCK,$dest_addr)

Funkcja connect() próbuje połączyć gniazdo zorientowane na połączenie ze wskazanym adresem docelowym. Gniazdo musiało być uprzednio utworzone za pomocą funkcji socket(), a upakowany adres docelowy za pomocą sockaddr_in() lub funkcji jej równoważnej. System automatycznie wybiera port, który będzie wykorzystany jako lokalny adres dla gniazda. W przypadku powodzenia funkcja connect() zwraca wartość logiczną *prawda*; w przeciwnym przypadku connect() zwraca wartość logiczną *fałsz*, zaś zmienna $! zostaje ustawiona na kod błędu systemu, wyjaśniający zaistniały problem. Nie można wywoływać connect() dla gniazda zorientowanego na połączenie więcej niż raz. Próba powtórnego wywołania tej funkcji spowoduje wystąpienie błędu EISCONN („Punkt końcowy transportu jest już połączony”).

$boolean = close (SOCK)

Funkcja close() współpracuje z gniazdami na tej samej zasadzie, jak w przypadku zwykłych uchwytów plików. Po jej wywołaniu gniazdo zostaje zamknięte dla petentów. Raz zamknięte nie może być dłużej używane do dokonywania weń zapisów czy odczytywania z niego danych. W przypadku powodzenia funkcja close() zwraca wartość logiczną *prawda*, w przeciwnym razie zwraca wpis undef i pozostawia komunikat o błędzie w zmiennej $!. Wpływ funkcji close() na przeciwległy koniec połączenia można porównać z zamykaniem potoku. Po zamknięciu gniazda wszystkie dokonywane z niego odczyty będą na przeciwległym końcu połączenia zwracać symbol końca pliku (EOF). Jakiekolwiek zapisy do gniazda spowodują wyjątek PIPE.

$boolean = shutdown (SOCK,$how)

Funkcja `shutdown()` jest bardziej precyzyjną odmianą funkcji `close()`, pozwalającą użytkownikowi na podjęcie decyzji, którą część dwukierunkowego połączenia należy zamknąć. Pierwszy argument tej funkcji to podłączone gniazdo. Argument drugi — `$how` — jest małą liczbą całkowitą, wskazującą, na którym końcu ma nastąpić zamknięcie połączenia. W tabeli 4.1 zebrane zostały wartości przyjmowane przez argument `$how`. Argument o wartości 0 zamyka gniazdo dla mających nastąpić odczytów, wartość 1 powoduje zamknięcie gniazda dla zapisów, 2 zaś zamyka gniazdo zarówno dla odczytów, jak i dla zapisów (podobnie jak dzieje się to w przypadku funkcji `close()`). Zwrócenie wartości niezerowej wskazuje na powodzenie funkcji `shutdown()`.

Tabela 4.1. *Wartości funkcji shutdown()*

Wartość HOW	Opis
0	zamyka gniazdo dla odczytu
1	zamyka gniazdo dla zapisu
2	całkowicie zamyka gniazdo

Oprócz zdolności do połowicznego zamykania gniazda, funkcja `shutdown()` ma jeszcze jedną przewagę nad `close()`. Jeśli proces wywołał na jakimś etapie funkcję `fork()`, w procesach potomnych mogą istnieć kopie uchwytu pliku gniazda. Próba zwykłego zamknięcia za pomocą `close()` jakiejś kopii gniazda w rzeczywistości nie zamknie samego gniazda aż do momentu, gdy zamknięte zostaną wszystkie jego kopie (zachowanie to dotyczy również uchwytów plików). W wyniku tego klient na przeciwległym końcu połączenia nie otrzyma informacji `EOF`, dopóki proces macierzysty i proces (lub procesy) potomny nie zamkną swoich kopii. W odróżnieniu od `close()`, funkcja `shutdown()` zamyka wszystkie kopie gniazda, wysyłając natychmiast znak `EOF`. Skorzystamy z tej zalety `shutdown()`w tej książce kilkakrotnie.

Serwer usługi echo w protokole TCP

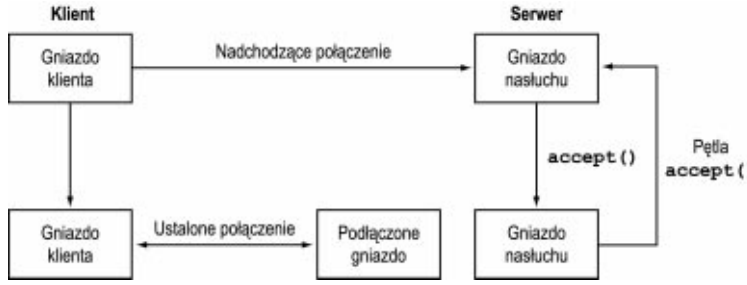
Przyjrzymy się teraz prostemu serwerowi TCP. W przeciwieństwie do klienta TCP, serwer zazwyczaj nie wywołuje funkcji `connect()`. Zamiast tego serwer TCP podąża za następującymi wytycznymi:

1. **Utwórz gniazdo**. To etap identyczny z odpowiednim etapem tworzenia klienta usługi echo (wiersz 12).
2. **Powiąż gniazdo z lokalnym adresem**. Program klienta może pozwolić systemowi operacyjnemu na wybranie adresu IP i numeru portu do wykorzystania przy wywołaniu funkcji `connect()`; adres serwera musi być jednak bardzo dobrze znany. Z tego powodu serwer musi wyraźnie powiązać gniazdo z lokalnym adresem IP i numerem portu. Jest to proces znany jako powiązanie (ang. *binding*). Funkcja towarzysząca temu procesowi to `bind()`.
3. **Oznacz gniazdo jako nasłuchujące**. Serwer wywołuje funkcję `listen()`, by poinformować system operacyjny, że gniazdo będzie wykorzystane do przyjmowania nadchodzących połączeń. Funkcja ta określa też liczbę

nadchodzących połączeń, które mogą oczekiwać w kolejce, zanim zostaną przyjęte przez serwer.

Gniazdo, które zostało oznaczone jako gotowe do przyjmowania nadchodzących połączeń jest nazywane gniazdem nasłuchu (ang. *listening socket*).

4. **Przyjmij nadchodzące połączenia**. Serwer wywołuje teraz — zazwyczaj w pętli — funkcję `accept()`. Przy każdym wywołaniu funkcja ta oczekuje na nadchodzące połączenie, a potem zwraca nowe, podłączone gniazdo, które jest przyłączone do gniazda równorzędnego, zdalnego partnera (rysunek 4.1). Operacja ta nie ma żadnego wpływu na gniazdo nasłuchu.



Rysunek 4.1. *Odbierając nadchodzące połączenie, funkcja accept() zwraca nowe gniazdo połączone z klientem*

5. **Wykonaj operacje wejścia-wyjścia na podłączonym gnieździe**. Serwer wykorzystuje podłączone gniazdo do komunikacji z równorzędnym, zdalnym partnerem (ang. *peer*). Po zakończonej pracy serwer zamyka podłączone gniazdo.

6. **Przyjmij więcej połączeń**. Wykorzystując gniazdo nasłuchu serwer może przyjąć (za pomocą funkcji `accept()`) dowolną liczbę połączeń. Po ich zakończeniu serwer zamknie — wykorzystując funkcję `close()` — gniazdo nasłuchu i zakończy pracę.

Nasz przykładowy serwer nosi nazwę *tcp_echo_serv1.pl*. Jest to nieco wypaczona wersja standardowego serwera usługi echo. Powtarza to wszystko, co zostało do niego wysłane, ale zamiast odesłać dane w pierwotnej formie, powtarza każdy wiersz wspak, zachowując bez zmian tylko znak nowego wiersza. Jeśli zatem zostanie przesłane do serwera pozdrowienie „Hello world!”, odesłane echo będzie następujące: „!dlrow olleH” (nie ma żadnych powodów dla takiego przetwarzania informacji, poza chęcią nieznacznego ubarwienia tego trochę nudnego przykładu).

Serwer ten może być użyty przez klienta z wydruku 4.1 lub wraz ze standardowym programem Telnet. Wydruk 4.2 pokazuje kod serwera.

Wydruk 4.2. *Skrypt tcp_echo_serv1.pl dostarcza sieciową usługę echa w protokole TCP*

```
0   #!/usr/bin/perl
1   # plik: tcp_echo_serv1.pl
2   # uzycie: tcp_echo_serv1.pl [port]

3   use strict;
4   use Socket;
5   use IO::Handle;
6   use constant MY_ECHO_PORT => 2007;
```

```
7   my ($bytes_out,$bytes_in) = (0,0);

8   my $port     = shift || MY_ECHO_PORT;
9   my $protocol = getprotobyname('tcp');

10   $SIG{'INT'} = sub {
11       print STDERR "bajty_wysłane = $bytes_out, bajty_odebrane = $bytes_in\n";
12       exit 0;
13   };

14   socket(SOCK, AF_INET, SOCK_STREAM, $protocol) or die "fiasko socket(): $!";
15   setsockopt(SOCK,SOL_SOCKET,SO_REUSEADDR,1)    or die "Nie można ustawić
                                                          SO_REUSADDR: $!" ;

16   my $my_addr = sockaddr_in($port,INADDR_ANY);
17   bind(SOCK,$my_addr)    or die "fiasko bind(): $!";
18   listen(SOCK,SOMAXCONN) or die "fiasko listen(): $!";

19   warn "czekam na połączenia nadchodzące do portu $port...\n";

20   while (1) {
21     next unless my $remote_addr = accept(SESSION,SOCK);
22     my ($port,$hisaddr) = sockaddr_in($remote_addr);
23     warn "Połączenie z [",inet_ntoa($hisaddr),",$port]\n";

24     SESSION->autoflush(1);
25     while (<SESSION>) {
26       $bytes_in  += length($_);
27       chomp;

28       my $msg_out = (scalar reverse $_) . "\n";
29       print SESSION $msg_out;
30       $bytes_out += length($msg_out);
31     }
32     warn "Połączenie z [",inet_ntoa($hisaddr),",$port] zakończono\n";
33     close SESSION;

34   }

35   close SOCK;
```

Wiersz 1 – 9: Ładuj moduły, inicjalizuj stałe i zmienne. Rozpoczynamy — podobnie jak w usłudze dla klienta — od wprowadzenia modułów `Socket` i `IO:Handle`. Określamy dla echa prywatny port o numerze 2007; nie będzie on pozostawał w konflikcie z żadnym istniejącym portem dla serwera usługi echo. Ustalamy — w znany już sposób — zmienne `$port` i `$protocol` (wiersze 1 – 8) i inicjalizujemy liczniki.

Wiersze 10 – 13: Zainstaluj procedurę obsługi przerwania `INT`. Musi istnieć sposób na przerwanie pracy serwera; z tego powodu należy zainstalować procedurę obsługi dla sygnału przerwania `INT`, wysyłanego z terminala, na którym użytkownik wciśnie kombinację klawiszy *CTRL+C*. Procedura obsługi sygnału `INT` drukuje zebraną statystykę zliczeń bajtów i na tym kończy działanie.

Wiersz 14: Utwórz gniazdo. Wykorzystując argumenty analogiczne do tych, jakie zastosowano w usłudze dla klienta (wydruk 4.1), wywołujemy funkcję `socket()`, by utworzyć gniazdo strumieniowe TCP.

Wiersz 15: Ustaw opcję gniazda `SO_REUSEADDR`. Kolejnym krokiem jest wywołanie `setsockopt()`w celu ustawienia wartości logicznej *prawda* dla opcji `SO_REUSEADDR` gniazda. Dzięki tej opcji możliwe jest natychmiastowe wstrzymanie pracy i ponowne natychmiastowe uruchomienie serwera. Gdyby tej opcji nie ustawiono, to w pewnych warunkach system mógłby nie zezwolić na ponowne powiązanie serwera z adresem lokalnym aż do chwili, kiedy stare połączenia dobiegłyby końca.

Wiersze 16 – 17: Powiąż gniazdo z adresem lokalnym. Wywołanie `bind()` przypisuje adres lokalny do gniazda. Adres ten jest tworzony przy użyciu funkcji `sockaddr_in()`, której zostaje przekazany jako port numer prywatnego portu dla echa i parametr `INADDR_ANY` jako adres IP. `INADDR_ANY` działa jako symbol wieloznaczny (ang. *wildcard*). Umożliwia to systemowi operacyjnemu przyjmowanie połączeń na każdym z adresów IP hosta, z adresem pętli zwrotnej i każdej sieciowej karty interfejsowej hosta włącznie.

Wiersz 18: Wywołaj funkcję `listen()`, **by przygotować gniazdo na przyjęcie nadchodzących połączeń**. Wywołanie funkcji `listen()` informuje system operacyjny, że gniazdo będzie wykorzystywane do przyjmowania nadchodzących połączeń. Funkcja ta przyjmuje dwa argumenty. Pierwszym jest gniazdo, drugim — liczba całkowita wskazująca maksymalną liczbę nadchodzących połączeń, jakie mogą oczekiwać w kolejce na przetworzenie. Często zdarzają się próby niemal jednoczesnego połączenia z gniazdem ze strony rozmaitych klientów; w takim przypadku drugi argument określa, jak duże mogą być zaległości w przyjmowaniu czekających na przetworzenie połączeń. W naszym przykładzie ustalamy tę wartość na maksymalną dopuszczalną przez system liczbę oczekujących połączeń, która jest określona za pomocą stałej `SOMAXCONN`. Stała ta jest zdefiniowana w module `Socket`.

Wiersze 19 – 34: Główna pętla. Większość kodu zajmuje główna pętla serwera, w której serwer oczekuje na nadchodzące połączenia; w niej także wykonywana jest ich obsługa.

Wiersz 21: Akceptuj nadchodzące połączenie**. Każde wykonanie pętli** wiąże się z wywołaniem funkcji `accept()`, wykorzystującej gniazdo nasłuchu jako swój drugi argument, nazwę nowego gniazda (`SESSION`) zaś jako argument pierwszy (właśnie taka, choć może wydawać się to dziwne, jest prawidłowa kolejność argumentów). Jeśli wywołanie funkcji `accept()` kończy się powodzeniem, funkcja zwraca jako wynik upakowany adres zdalnego gniazda, zaś za pośrednictwem argumentu `SESSION` zostaje zwrócone podłączone gniazdo.

Wiersze 22 – 23: Rozpakuj adres klienta. Wywołujemy `sockaddr_in()` w kontekście listowym, by rozpakować zwrócony przez funkcję `accept()` adres klienta. Składniki adresu otrzymane po rozpakowaniu to port klienta i adres IP. Adres drukujemy na standardowym wyjściu błędu. W rzeczywistej aplikacji taka informacja mogłaby być zapisana w elektronicznie datowanym pliku rejestru zdarzeń (ang. *time-stamped log file*).

Wiersze 24 – 33: Obsłuż połączenie**. Ten fragment** kodu obsługuje komunikację z klientem wykorzystującym podłączone gniazdo. Najpierw umieszczamy gniazdo `SESSION` w trybie automatycznego opróżniania, aby zapobiec kłopotom związanym z buforowaniem. Teraz można odczytywać za każdym razem po jednym wierszu z gniazda, używając operatora <>, lub zapisać tekst w wierszu na wspak i odsyłać go do klienta, wykorzystując funkcję `print()`. Trwa to do momentu, aż `<SESSION>` zwróci wpis

undef, co będzie znaczyło, że równorzędny zdalny parter zamknął połączenie po swojej stronie. Zamykamy gniazdo SESSION, drukujemy komunikat o stanie i wracamy do funkcji accept() w oczekiwaniu na kolejne nadchodzące połączenie.

Wiersz 35: Uporządkuj. Po zakończeniu głównej pętli „porządkujemy" system, zamykając otwarte gniazdo nasłuchu. Ta część kodu nigdy jednak nie następuje, gdyż serwer jest zaprojektowany tak, by jego pracę kończył klawisz przerwania.

Nasz przykładowy serwer uruchomiony z wiersza poleceń drukuje komunikat czekam na połączenia nadchodzące do portu, a potem wstrzymuje działanie do chwili, gdy uzyska jakieś połączenie. W sesji, której zapis teraz przedstawimy, widać dwa połączenia — jedno od klienta lokalnego spod adresu 127.0.0.1 na pętli zwrotnej, drugie zaś od klienta z adresem 192.168.3.2. Przerwanie pracy serwera pozwoli na zapoznanie się ze statystyką, wydrukowaną przez procedurę obsługi przerwania INT.

```
% tcp_echo_serv1_pl
czekam na polaczenia nadchodzace do portu 2007...
Połączenie z [127.0.0.1,2865]
Połączenie z [127.0.0.1,2865] zakończono
Połączenie z [192.168.3.2,2901]
Połączenie z [192.168.3.2,2901] zakończono
^C
bajty_wysłane = 26, bajty_odebrane = 26
```

Procedura obsługi INT użyta w tym serwerze sprzeciwia się zaleceniom z rozdziału 2., mówiącym o tym, że procedury obsługi sygnałów nie obsługują żadnych operacji wejścia-wyjścia. Ponadto wywołanie funkcji exit() z wnętrza procedury obsługi niesie ryzyko powstania wyjątku krytycznego w trakcie zamykania systemu w komputerach z systemem operacyjnym Windows. Bezpieczniejszy sposób zamykania serwera poznamy w rozdziale 10.

Funkcje gniazda związane z połączeniami nadchodzącymi

Trzy kolejne, niezbędne funkcje są związane z obsługą nadchodzących połączeń w serwerach.

$boolean = bind(SOCK,$my_addr)

Funkcja bind() wiąże adres lokalny z gniazdem, zwracając — w przypadku powodzenia — wartość logiczną *prawda* lub *fałsz* w razie niepowodzenia. Gniazdo musi być uprzednio utworzone za pomocą funkcji socket(), a upakowany adres wygenerowany za pomocą sockaddr_in() lub funkcji jej równoważnej. W części adresu określającej port można umieścić numer jednego z portów nieużywanych w systemie. Adres IP może być adresem jednego z interfejsów sieciowych hosta, adresem pętli zwrotnej albo symbolem wieloznacznym — INADDR_ ANY. W systemach UNIX do powiązania z zarezerwowanymi portami o numerach niższych niż 1024 wymagane są przywileje superużytkownika (użytkownika root). Próba dokonania powiązania bez takich przywilejów spowoduje zwrócenie wartości undef i ustawienie $! w pozycji błędu EACCES ("Permission denied" — „Brak pozwolenia"). Funkcja bind() jest zazwyczaj wywoływana w serwerach w celu powiązania nowo utworzonego gniazda z dobrze znanym portem, jednak klient może wywołać tę funkcję także wtedy, gdy ma zamiar określić lokalny port i (lub) interfejs sieciowy.

$boolean = listen(SOCK,$max_queue)

Funkcja `listen()` informuje system operacyjny, że gniazdo będzie użyte do przyjmowania nadchodzących połączeń. Dwoma jej argumentami są uchwyt pliku gniazda, które musiało być uprzednio utworzone przy funkcji użyciu `socket()`, oraz wartość całkowita, wskazująca na liczbę nadchodzących połączeń, które mogą oczekiwać w kolejce do przetworzenia. Maksymalna długość kolejki jest zależna od systemu. Określenie wartości większej niż akceptowana przez dany system spowoduje, że funkcja `listen()` automatycznie zredukuje zawyżoną wartość do dopuszczalnej dla systemu wartości maksymalnej. Moduł `Socket` eksportuje stałą `SOMAXCONN` dla określenia tej maksymalnej wartości. W przypadku powodzenia funkcja `listen()` zwraca wartość logiczną *prawda* i oznacza gniazdo jako nasłuchujące. W przeciwnym razie zwraca wpis `undef` i ustawia `$!` na odpowiednią wartość błędu.

$remote_addr = accept(CONNECTED_SOCKET,LISTEN_SOCKET)

Jeśli gniazdo jest oznaczone jako nasłuchujące, należy wywołać funkcję `accept()` do przyjmowania nadchodzących połączeń. Funkcja ta przyjmuje dwa argumenty: `CONNECTED_SOCKET` — nazwę uchwytu pliku przeznaczonego dla nowo podłączonego gniazda i `LISTEN_SOCKET` — nazwę gniazda nasłuchu. W przypadku powodzenia jako wynik funkcji zostanie zwrócony upakowany adres zdalnego hosta, a argument `CONNECTED_SOCKET` zostanie skojarzony z nadchodzącym połączeniem. Po wykonaniu funkcji `accept()` do komunikacji z równorzędnym zdalnym klientem będzie użyty uchwyt `CONNECTED_SOCKET`. Nie ma potrzeby tworzenia tego uchwytu „na zapas". Jeśli wydaje się to Czytelnikowi trochę niejasne, niech wyobrazi sobie funkcję `accept()` jako specjalną odmianę funkcji `open()`, w której nazwę pliku zastępuje nazwa gniazda nasłuchu `LISTEN_SOCKET`.

Jeśli na przyjęcie nie czeka żadne połączenie, funkcja `accept()` będzie zablokowana do chwili nadejścia jakiegoś połączenia. Jeśli zbyt wiele aplikacji klienckich łączy się z serwerem szybciej, niż skrypt jest w stanie wywoływać funkcję `accept()`, będą one musiały czekać w kolejce, której parametry określa wywołanie `listen()`. Funkcja `accept()` zwraca wartość niezdefiniowaną `undef`, jeśli zaistnieje jakikolwiek z licznych warunków wystąpienia błędów i ustawia `$!` na odpowiedni komunikat o błędzie.

$my_addr = getsockname(SOCK)
$remote_addr = getpeername(SOCK)

Gdy zaistnieje potrzeba odzyskania lokalnego lub zdalnego adresu skojarzonego z gniazdem, można wykorzystać do tego celu funkcję `getsockname()` lub `getpeername()`. Funkcja `getsockname()` zwraca upakowany adres binarny lokalnego gniazda oraz wartość `undef`, jeśli gniazdo nie jest powiązane. Funkcja `getpeername()` zachowuje się podobnie — jedyna różnica polega na tym, że zwraca adres zdalnego gniazda i wartość niezdefiniowaną `undef`, jeśli to gniazdo nie jest podłączone. Obydwie funkcje zwracają adresy, które muszą być rozpakowane przy użyciu funkcji `sockaddr_in()` — tak, jak pokazuje następujący przykład:

```
If ($remote_addr = getpeername(SOCK)) {
   my ($port,$ip) = sockaddr_in($remote_addr);
   my $host = gethostbyaddr($ip,AF_INEF);
   print "Gniazdo jest podłączone do $host na porcie $port\n";
}
```

Ograniczenia skryptu tcp_echo_serv1.pl

Skrypt *tcp_echo_serv1.pl* pracuje zgodnie z intencją jego autora, niemniej jednak posiada kilka wad, które omówimy w kolejnych rozdziałach. Do jego niedoskonałości należy zaliczyć:

1. *Brak obsługi nadchodzących połączeń wielokrotnych*. Jest to z całą pewnością największy problem. Serwer może przyjąć jednorazowo tylko jedno połączenie. W czasie, gdy jest zajęty obsługą przyjętego połączenia, wszystkie inne żądania połączenia będą oczekiwać na swoją kolej, aż zakończy się bieżące połączenie i pętla główna ponownie wywoła funkcję `accept()`. Gdy liczba klientów oczekujących na swą kolej przekroczy wartość podaną przez `listen()`, wszystkie nowe połączenia zostaną odrzucone.

   Aby ominąć ten problem, serwer musiałby przetwarzać współbieżnie wątki lub procesy albo zwielokrotniać swoje operacje wejścia-wyjścia. Takie techniki będą szczegółowo omówione w III części tej książki.

2. *Serwer pozostaje na pierwszym planie*. Po uruchomieniu serwer pozostaje na pierwszym planie i każdy sygnał docierający z klawiatury (jak choćby kombinacja klawiszy *Ctrl+C*) może przerwać jego pracę. Dla serwerów pracujących w trybie długotrwałym konieczne będzie uniezależnienie ich od poleceń wprowadzanych z klawiatury i przeniesienie z pierwszego planu do procesu działającego w tle. Techniki niezbędne do wykonania tego zadania opisane są w rozdziale 10., „Serwery współbieżne oraz demon inetd".

3. *Zapis w rejestrze zdarzeń serwera jest uproszczony*. Serwer zapisuje informację o stanie na strumień standardowego wyjścia błędu. Jednakże solidny serwer będzie uruchomiony jako proces w tle (ang. *background process*), a zatem nie powinno się określać dla niego standardowego wyjścia błędu do zapisu. Serwer powinien dopisywać wpisy rejestrujące zdarzenia do pliku albo wykorzystać do tego celu oferowane przez sam system operacyjny narzędzia pozwalające dokonywać rejestracji zdarzeń w dzienniku systemowym. Techniki zapisywania w rejestrze zdarzeń systemowych są omówione w rozdziale 14., „Ochrona serwerów".

Regulacja ustawień opcji gniazd

Gniazda dysponują zestawem opcji, które sterują rozmaitymi aspektami ich działań. Można — między innymi — regulować rozmiary buforów, używanych do wysyłania i przyjmowania danych, dostosowywać wartości limitów czasowych dla wysyłania i przyjmowania danych, istnieje ponadto możliwość zadecydowania, czy gniazdo może być wykorzystane do odbierania transmisji rozgłaszania (ang. *broadcast transmissions*).

Opcje ustawione domyślnie sprawdzają się w większości przypadków. Niekiedy jednak zachodzi potrzeba regulacji niektórych z nich w celu udoskonalenia jakiejś aplikacji lub uaktywnienia opcjonalnych cech protokołu TCP/IP. Najczęściej używaną opcją jest `SO_REUSEADDR`, uruchamiana powszechnie w aplikacjach serwera.

Opcje gniazda mogą być sprawdzone lub zmienione za pomocą wbudowanych funkcji Perla — `getsockopt()` oraz `setsockopt()`.

$value = getsockopt(SOCK,$level,$option_name);
$boolean = setsockopt(SOCK,$level,$option_name,$option_value);

Funkcje `getsockopt()` oraz `setsockopt()` umożliwiają sprawdzenie i zmianę opcji gniazda. Pierwszym ich argumentem jest uchwyt pliku `SOCK` dla uprzednio utworzonego gniazda. Argument drugi — `$level` — wskazuje poziom stosu sieciowego, na którym ma być wykonana operacja. Najczęściej wykorzystuje się stałą `SOL_SOCKET`, oznaczającą, że operacje dokonywane są na samym gnieździe. Niekiedy jednak funkcje `getsockopt()` i `setsockopt()` są wykorzystywane do regulacji opcji w protokołach TCP i UDP. W takiej sytuacji używa się numeru protokołu zwróconego przez funkcję `getprotobyname()`. Wartość trzeciego argumentu — `$option_name` — jest liczbą całkowitą, wybraną z obszernej listy możliwych stałych. Ostatni argument — `$option_value` — jest wartością, która ma przyjąć opcja. W przypadkach, w których wartość opcji nie znajduje zastosowania, można podać wartość niezdefiniowaną `undef`. W przypadku powodzenia `getsockopt()` zwraca wartość żądanej opcji, w razie niepowodzenia — wartość `undef`. Funkcja `setsockopt()` zwraca wartość logiczną *prawda*, gdy opcja została pomyślnie ustawiona; w przeciwnym przypadku zwraca wartość `undef`.

Wartość opcji jest niejednokrotnie znacznikiem logicznym (boolowskim), wskazującym, czy opcja powinna być aktywna, czy też nie. W takiej sytuacji do ustawienia i zmiany wartości nie jest potrzebny żaden specjalny kod. W tym przykładzie zademonstrowano, jak ustawić wartość opcji `SO_BROADCAST` dla wartości logicznej *prawda* (rozgłaszanie jest omawiane w rozdziale 20.):

```
setsockopt(SOCK,SOL_SOCKET,SO_BROADCAST,1);
```

Tutaj przedstawiony jest sposób odzyskiwania bieżącej wartości znacznika:

```
my $reuse = getsockopt(SOCK,SOL_SOCKET,SO_BROADCAST);
```

Kilka opcji działa na liczbach całkowitych lub innych, rzadko używanych typach danych, takich jak choćby struktury `timeval` języka C. W takim przypadku, przed przekazaniem do funkcji `setsockopt()` należy skompresować wartości do postaci binarnej i rozpakować je po wywołaniu `getsockopt()`. Dla zilustrowania tego problemu przedstawimy teraz sposób odzyskiwania maksymalnego rozmiaru buforu, którego używa gniazdo do przechowywania danych. Opcja `SO_SNDBUF` działa na skompresowanej liczbie całkowitej (format `I`):

```
$send_buffer_size = unpack("I",getsockopt($sock,SOL_SOCKET,SO_SNDBUF));
```

Typowe opcje gniazda

W tabeli 4.2 zebrane są typowe opcje gniazd, używane w programowaniu sieciowym. Stałe podane w tabeli są standardowo importowane przy ładowaniu modułu `Socket`.

Oto bardziej szczegółowy opis tych opcji:

Opcja `SO_REUSEADDR` — umożliwia ponowne powiązanie gniazda TCP z będącym w użyciu adresem lokalnym. Opcja ta przyjmuje argument logiczny (boolowski) wskazujący, czy należy uaktywnić ponowne wykorzystanie adresu. Więcej informacji na ten temat zamieszczono w dalszej części tego rozdziału, w punkcie „Opcja SO_ REUSEADDR dla gniazda”.

Tabela 4.2. *Typowe opcje gniazd*

Opcja	Opis
SO_REUSEADDR	Uaktywnia ponowne użycie adresu lokalnego
SO_KEEPALIVE	Uaktywnia transmisję okresowych komunikatów sprawdzających aktywność (ang. *keepalive messages*)
SO_LINGER	Opóźnia zamknięcie gniazda, jeśli są jeszcze dane do wysłania
SO_BROADCAST	Umożliwia gniazdu wysyłanie komunikatów na adres rozgłaszania
SO_OOBINLINE	Umożliwia wstawianie pilnych danych do strumienia w celu przesłania ich poza kolejnością
SO_SNDLOWAT	Pobiera lub ustawia poziom minimalny (ang. *low water mark*) dla rozmiaru buforu wyjściowego
SO_RECVLOWAT	Pobiera lub ustawia poziom minimalny dla rozmiaru buforu wejścia
SO_TYPE	Pobiera typ gniazda (w trybie tylko do odczytu)
SO_ERROR	Pobiera oraz usuwa ostatni błąd w gnieździe (tylko do odczytu)

Opcja SO_KEEPALIVE — opcja ta, o wartości logicznej *prawda*, nakazuje, by podłączone gniazdo okresowo przesyłało komunikaty do równorzędnego zdalnego partnera (ang. *peer*). Jeśli zdalny host nie odpowiada na przesłaną wiadomość, to proces przy kolejnej próbie zapisu do gniazda otrzyma sygnał PIPE. Odstęp czasowy w wysyłaniu komunikatów sprawdzających aktywność połączenia (ang. *keepalive messages*) nie może być ustalony w sposób dający się przenosić pomiędzy systemami. Wartość ta jest zróżnicowana w zależności od systemu operacyjnego (przykładowo, dla systemu Linux ten odstęp czasowy wynosi 45 sekund).

Opcja SO_LINGER — nadzoruje wydarzenia zachodzące przy próbie zamknięcia gniazda TCP, w którym ciągle czekają na wysłanie jakieś dane. Zwykle funkcja close() natychmiast kończy działanie, a system operacyjny stara się wysłać w tle pozostałe dane. Poprzez ustawienie opcji SO_LINGER można również zablokować funkcję close() w trakcie wywołania aż do chwili, gdy wszystkie dane zostaną wysłane. Umożliwia to sprawdzenie, czy wartość zwrócona przez close() informuje o pomyślnie zakończonym zadaniu.

W przeciwieństwie do innych opcji gniazda, SO_LINGER działa na skompresowanym typie danych — tak zwanej strukturze *linger*. Struktura ta składa się z dwóch liczb całkowitych — znacznika wskazującego, czy opcja SO_LINGER powinna być aktywna oraz z wartości ograniczenia czasowego (ang. *timeout*), podającego maksymalną liczbę sekund, o jakie close() powinna opóźnić swoje zakończenie. Struktura *linger* powinna być skompresowana i rozpakowana przy użyciu formatu II:

```
$linger = pack("II",$flag,$timeout);
```

Na przykład, aby gniazdo opóźniało swe zamknięcie przez 120 sekund, należałoby wpisać:

```
setsockopt(SOCK,SOL_SOCKET, SO_LINGER, pack("II",1,120))
                    or die "Nie można ustawić SO_LINGER: $!";
```

Opcji SO_BROADCAST można użyć poprawnie tylko w stosunku do gniazd UDP. Jeśli opcja ta ma wartość logiczną *prawda*, funkcja send() może być użyta do wysyłania pakietów na adres rozgłaszania (ang. *broadcast address*) w celu dostarczenia ich do wszystkich hostów w lokalnej podsieci. Zagadnienia związane z rozgłaszaniem omówione są w rozdziale 20.

Znacznik opcji SO_OOBINLINE steruje obsługą pilnych danych, czyli informacji obsługiwanych poza kolejnością (ang. *out-of-band information*). Dzięki temu równorzędny zdalny partner zostaje zaalarmowany o obecności danych o wysokim priorytecie. W rozdziale 17. opisano to nieco dokładniej.

Opcje SO_SNDLOWAT oraz SO_RCVLOWAT ustawiają poziom minimalny dla rozmiaru buforów — odpowiednio wyjścia i wejścia. Znaczenie tych opcji jest szerzej omówione w rozdziale 13., zatytułowanym „Nieblokujące operacje wejścia-wyjścia". Obie opcje są liczbami całkowitymi i muszą być kompresowane i rozpakowywane z użyciem formatu kompresowania I.

Opcja SO_TYPE jest opcją przeznaczoną tylko do odczytu. Zwraca typ gniazda, na przykład SOCK_STREAM. Przed użyciem trzeba rozpakować tę wartość z pomocą formatu I. Metoda sockopt() modułu IO::Socket, omówiona w rozdziale 5., dokonuje automatycznej konwersji.

Ostatnia z typowych opcji gniazda — SO_ERROR — jest także opcją przeznaczoną tylko do odczytu; zwraca kod błędu (jeśli wystąpił błąd) dla ostatniej operacji. Używana jest dla pewnych operacji asynchronicznych, takich jak połączenia nieblokujące (zobacz rozdział 13.). Błąd jest usuwany po jego odczytaniu. Tak jak w poprzednich przypadkach, użytkownicy getsockopt() muszą rozpakować tę wartość przed jej użyciem za pomocą formatu I. Automatycznie dokonuje tego moduł IO::Socket.

Opcja SO_REUSEADDR dla gniazda

Wielu programistów zapragnie aktywować znacznik opcji SO_REUSEADDR w aplikacjach serwera. Znacznik ten umożliwia serwerowi powtórne powiązanie z adresem, który jest już w użyciu. To z kolei pozwala serwerowi na ponowne uruchomienie, następujące natychmiast po krachu serwera lub po przerwaniu jego pracy. Bez tej opcji wywołanie bind() nie powiedzie się, dopóki wszystkie nawiązane wcześniej połączenia nie wyczerpią swoich limitów czasowych — czyli nawet przez kilka minut.

Aktywacja opcji SO_REUSEADDR polega na wstawieniu następującego wiersza kodu po wywołaniu funkcji socket(), a przed wywołaniem funkcji bind():

```
setsockopt(SOCK,SOL_SOCKET,SO_REUSEADDR,1) or die "setsockopt: $";
```

Pewnym mankamentem ustawienia opcji SO_REUSEADDR jest wystąpienie możliwości dwukrotnego uruchomienia serwera. W takim przypadku obydwa procesy będą mogły wiązać się z tym samym adresem bez powodowania błędu, a następnie będą rywalizowały o nadchodzące połączenia, co będzie prowadzić do mylących wyników. Serwery, które zostaną opracowane w kolejnych rozdziałach (na przykład w rozdziałach 10., 14. i 15.), omijają taką ewentualność, tworząc przy uruchomieniu programu plik i usuwając go przy zakończeniu programu. Serwer odmawia rozpoczęcia pracy, gdy dostrzega istnienie takiego pliku.

Bez względu na ustawienia opcji SO_REUSEADDR, system operacyjny nie pozwala, by adres gniazda powiązany przez proces jakiegoś użytkownika był powiązany z procesem innego użytkownika.

Funkcje fcntl() i ioctl()

Oprócz opcji gniazda do regulacji ustawień licznych atrybutów mogą być wykorzystane funkcje fcntl() i ioctl(). Funkcja fcntl() jest omówiona w rozdziale 13., w którym wykorzystana jest do włączenia nieblokującej operacji wejścia-wyjścia oraz w rozdziale 17., w którym użyto jej do ustawienia właściciela gniazda tak, by otrzymywał on sygnał URG wtedy, gdy gniazdo otrzymuje pilne dane TCP.

Także funkcja ioctl() pojawia się w rozdziale 17., w którym jest wykorzystana do implementacji funkcji sockatmark(), obsługującej pilne dane. Ponadto napotkamy ją w rozdziale 21., gdzie utworzony zostanie cały wachlarz funkcji sprawdzających i modyfikujących adresy IP przypisane interfejsom sieciowym.

Inne funkcje odnoszące się do gniazd

Do poznanych już funkcji związanych z gniazdami doliczyć należy trzy kolejne wbudowane funkcje Perla: send(), recv() oraz socketpair(). Dwie pierwsze będą wykorzystane w późniejszych rozdziałach tej książki, przy omawianiu pilnych danych TCP (rozdział 17.) i protokołu UDP (rozdziały 17 – 20).

$bytes = send(SOCK,$data,$flags[,$destination])

Funkcja send() używa gniazda wskazanego przez pierwszy argument SOCK, by dostarczyć dane, wskazane przez argument $data, na adres docelowy, określony przez $destination. Jeśli dane pomyślnie zostały ustawione w kolejce do przesłania, funkcja send() zwróci liczbę wysłanych bajtów. W przeciwnym przypadku zwraca wartość nieokreśloną — undef. Argument trzeci — $flags — może mieć wartość 0, wartość jednej z dwóch opcji wybranych z tabeli 4.3 lub może być określony jako wynik działania operatora bitowej alternatywy dla tych dwóch opcji. Znacznik MSG_OOB zostanie omówiony szczegółowo w rozdziale 17. Znacznik MSG_DONTROUTE jest używany w programach określających marszruty (ang. *routing programs*) oraz w programach diagnostycznych i nie będzie omawiany w tej książce. Wyrażeniem zgody na standardowe zachowanie funkcji send() będzie przekazanie 0 jako wartości argumentu $flags. Jeżeli gniazdo jest podłączonym gniazdem TCP, argument $destination nie powinien być określony, a funkcja send() będzie w pewnym stopniu odpowiadała funkcji syswrite(). W przypadku gniazd UDP adres docelowy może być zmieniany przy każdym wywołaniu funkcji send().

Tabela 4.3. *Znaczniki send()*

Opcja	Opis
MSG_OOB	Przekaż bajt pilnych danych na gniazdo TCP
MSG_DONTROUTE	Omiń tablice marszrut

$address = recv(SOCK,$buffer,$length,$flags)

Funkcja `recv()` przyjmuje ze wskazanego gniazda co najwyżej `$length` bajtów i umieszcza je w zmiennej skalarnej `$buffer`. Zmienna rośnie lub kurczy się do wielkości odpowiadającej faktycznie przeczytanej liczbie bajtów danych. Argument `$flags` ma znaczenie analogiczne do odpowiadającego mu argumentu w funkcji `send()` i powinien być z reguły ustawiony na wartość 0. W przypadku powodzenia funkcja `recv()` zwraca skompresowany adres gniazda nadawcy wiadomości. W razie błędu funkcja zwraca wartość nieokreśloną `undef` i ustawia odpowiednio zmienną `$!`. Gdy funkcja `recv()` jest wywołana na podłączonym gnieździe TCP, działa podobnie do funkcji `sysread()`, z tą różnicą, że zwraca adres równorzędnego zdalnego partnera. Przydatność funkcji `recv()` sprawdza się zwłaszcza przy przyjmowaniu datagramów w transmisji UDP.

$boolean = socketpair(SOCK_A,SOCK_B,$type,$protocol)

Funkcja `socketpair()` tworzy parę nie nazwanych gniazd połączonych swymi zakończeniami. Argumenty `$domain`, `$type` oraz `$protocol` odpowiadają analogicznym argumentom z funkcji `socket()`. W przypadku powodzenia funkcja `socketpair()` zwraca wartość logiczną *prawda* i otwiera gniazda uchwytu `SOCK_A` oraz `SOCK_B`.

Funkcja `socketpair()` przypomina funkcję `pipe()` z rozdziału 2., przy czym w tym przypadku połączenie jest dwukierunkowe. Zazwyczaj skrypt tworzy parę gniazd, a następnie rozwidla się za pomocą funkcji `fork()` na proces macierzysty, zamykający jedno gniazdo i proces potomny, zamykający drugie. Oba gniazda mogą być następnie użyte do dwukierunkowej komunikacji pomiędzy procesem macierzystym i potomnym.

Podczas gdy funkcja `socketpair()` jest z reguły używana w protokołach INET, w rzeczywistości większość systemów obsługuje ją tylko przy tworzeniu gniazd domeny UNIX. Oto schemat kodu tej funkcji:

```
socketpair(SOCK1,SOCK2,AF_UNIX,SOCK_STREAM,PF_UNSPEC) or die $!;
```

W rozdziale 22. zostaną przedstawione przykłady użycia gniazd domeny UNIX.

Stałe końca wiersza eksportowane przez moduł gniazda

Moduł `Socket`, jak już wiemy, wykorzystuje stałe do budowy gniazd i ustalania połączeń wychodzących, ale — o czym właśnie się przekonamy — może również eksportować stałe i zmienne, wykorzystywane w odniesieniu do zorientowanych tekstowo serwerów sieciowych.

Jak widzieliśmy w rozdziale 2., różne systemy operacyjne w inny sposób interpretują budowę końca wiersza w pliku tekstowym. Niektóre systemy używają znaku powrotu karetki (ang. *carriage return*, `CR`), inne znaku przesunięcia wiersza (ang. *linefeed*, `LF`), a jeszcze inne obydwu znaków, użytych łącznie (`CRLF`). Dodatkowe trudności wprowadzają znaki `\r` i `\n` — sekwencje sterujące z lewym ukośnikiem w Perlu, które w zależności od lokalnego systemu operacyjnego i jego sposobu interpretowania końca wiersza są tłumaczone na różne znaki ASCII.

Większość zorientowanych tekstowo usług sieciowych — choć nie jest to „sztywną" zasadą — kończy wiersze tekstu sekwencją CRLF, czyli — w zapisie ósemkowym — \015\012. Przy wykonywaniu zorientowanych wierszowo odczytów z takich serwerów należy ustawić separator pola wprowadzenia zapisu (ang. *input record separator*) — zmienną globalną $/ — tak, by przyjmował wartości \015\012 (ale nie \r\n, bo w takiej postaci nie można go przenosić między platformami systemowymi). Moduł Socket upraszcza te czynności, eksportując dodatkowo kilka stałych, definiujących typowe zakończenia wierszy (zobacz tabelę 4.4). Ponadto, by ułatwić interpolację tych sekwencji do postaci łańcuchowej, moduł Socket eksportuje zmienne $CRLF, $CR oraz $LF.

Tabela 4.4. *Stałe eksportowane przez moduł Socket*

Nazwa	Opis
CRLF	Stała zawierająca sekwencję CRLF
CR	Stała zawierająca znak CR
LF	Stała zawierająca znak LF

Symbole te nie są eksportowane standardowo; muszą być wprowadzone przy użyciu dyrektywy use — albo pojedynczo, albo poprzez zaimportowanie znacznika :crlf. W tym drugim przypadku, by pobrać równocześnie domyślne stałe odnoszące się do gniazd, potrzebne będzie zapewne również zaimportowanie znacznika :DEFAULT:

```
use Socket qw(:DEFAULT :crlf);
```

Wyjątkowe sytuacje podczas komunikacji

Protokół TCP jest niezwykle solidny w konfrontacji z nienajlepszymi warunkami sieciowymi. Może przetrwać powolne połączenia, niepewne rutery, przejściowe sieciowe przestoje, mnóstwo rozmaitych błędów konfiguracyjnych i wciąż jest zdolny dostarczyć zwarty, wolny od błędów strumień danych.

TCP nie może jednak przezwyciężyć wszystkich trudności. W tym podrozdziale zostaną omówione krótko typowe wyjątki, a także niektóre pospolite błędy programowania.

Wyjątki podczas wywołania connect()

Wywołania funkcji connect() sprzyjają pojawieniu się różnych typowych błędów.

1. *Zdalny host jest gotowy, ale żaden serwer nie nasłuchuje, gdy klient stara się uzyskać połączenie*. Klient próbuje połączyć się ze zdalnym hostem, ale żaden serwer nie prowadzi nasłuchu wskazanego portu. Funkcja połączenia connect() przerywa wykonanie, wyświetlając informację o błędzie ECONNREFUSED („Connection refused" — „Połączenie odrzucone").

2. *Zdalny host nie jest gotowy, gdy klient stara się uzyskać połączenie*. Klient próbuje połączyć się ze zdalnym hostem, ale ten nie pracuje (jest uszkodzony lub niedostępny). W takim przypadku funkcja `connect()` zostaje zablokowana na czas ograniczony, po którym następuje komunikat o błędzie `ETIMEDOUT` („`Connection timed out`" — „Minął czas połączenia"). Protokół TCP potrafi obsłużyć powolne połączenia sieciowe, zatem połączenia mogą nie wygasać ze względu na długie limity czasowe przez wiele minut.

3. *Sieć ma błędy konfiguracyjne*. Klient próbuje połączyć się ze zdalnym hostem, ale system operacyjny nie może poradzić sobie z wyborem marszruty dla przekazania wiadomości do żądanego celu ze względu na lokalne błędy konfiguracyjne albo niepowodzenie routera, jakie zaszło gdzieś na linii połączenia. W takim przypadku funkcja `connect()`kończy się niepowodzeniem i wyświetleniem komunikatu o błędzie `ENETUNREACH` („`Network is unreachable`" — „Sieć jest niedostępna").

4. *Błąd programisty*. Liczne błędy są spowodowane pospolitymi pomyłkami, popełnionymi podczas programowania. Na przykład próba wywołania funkcji `connect()` z uchwytem pliku, a nie gniazdem, spowoduje wystąpienie błędu `ENOTSOCK` („`Socket operation on non-socket`" — „Operacja dla gniazda nie wykonana na gnieździe"). Próba odwołania do `connect()` dla gniazda, które jest już podłączone, spowoduje wystąpienie błędu `EISCONN` („`Transport endpoint is already connected`" — „Punkt końcowy transportu jest już podłączony").

Komunikat o błędzie `ENOTSOCK` może być również zwrócony przez inne wywołania gniazda, takie jak `bind()`, `listen()`, `accept()` oraz `sockopt()`.

Wyjątki podczas operacji odczytu i zapisu

Po ustaleniu połączenia ciągle jest możliwe wystąpienie błędów. Jest niemal całkiem pewne, że podczas pracy z programami sieciowymi napotkamy następujące błędy:

1. *Następuje krach programu serwera w trakcie połączenia z klientem*. Jeśli następuje krach programu serwera podczas sesji komunikacyjnej, system operacyjny zamknie gniazdo. Z punktu widzenia klienta jest to taka sama sytuacja, jak celowe zamknięcie połączenia przez zdalny program na jego końcu gniazda.

   Przy odczytach, gdy po raz kolejny wywołana jest funkcja `read()` albo `sysread()`, pojawia się znak `EOF` („Koniec pliku"). Przy zapisach pojawia się wyjątek `PIPE`, dokładnie tak, jak to miało miejsce w przykładach z rozdziału 2. dotyczących potoków. Jeśli `PIPE` zostanie przechwycony i obsłużony, funkcja `print()` lub `syswrite()` zwróci wartość logiczną *fałsz* i zmienna `$!` zostanie ustawiona na wartość `EPIPE` (`Broken pipe` — „Przerwany potok"). W przeciwnym przypadku program zakończy się sygnałem `PIPE`.

2. *Następuje krach serwera hosta podczas nawiązanego połączenia*. Jeśli następuje krach hosta w trakcie aktywnego połączenia TCP, system operacyjny nie ma szansy na łagodne zakończenie połączenia. Po stronie użytkownika system operacyjny nie potrafi rozróżnić hosta nieczynnego od takiego, który zwyczajnie

natrafił na bardzo długi przestój w sieci. Host użytkownika będzie retransmitował pakiety IP w nadziei, że zdalny host pojawi się ponownie. Użytkownik ze swojej perspektywy dojrzy zablokowane na nieokreślony czas bieżące wywołania odczytów i zapisów.

Po jakimś czasie, gdy zdalny host znów się uaktywni, otrzyma jeden z pakietów retransmitowanych przez lokalny host. Nie potrafiąc zinterpretować tej sytuacji, zdalny host będzie przesyłać komunikat zerowania niskiego poziomu, informujący host lokalny o odrzuceniu połączenia. Na tym etapie połączenie zostaje przerwane, a program użytkownika — w zależności od wykonywanej operacji — uzyskuje informację albo o końcu pliku (`EOF`), albo o błędzie potoku.

Metodą ominięcia blokowania na nieokreślony czas jest ustawienie dla gniazda opcji `SO_KEEPALIVE`; wówczas połączenie wygasa w razie braku odpowiedzi w ciągu pewnego czasu, zaś gniazdo zostaje zamknięte. Wartość ograniczenia czasowego na podtrzymywanie aktywności połączenia jest względnie długa (dochodzi w niektórych przypadkach nawet do kilku minut) i nie można jej zmienić.

3. *Sieć przestaje działać podczas nawiązanego połączenia*. Jeśli w trakcie nawiązanego połączenia ruter lub jakiś segment sieci przestaje działać i z tego powodu zdalny host przestaje być dostępny, to bieżąca operacja wejścia-wyjścia zostaje zablokowana do momentu odzyskania utraconego połączenia. Jednak w takiej sytuacji, gdy zostaje przywrócone normalne działanie sieci, dalsze połączenie odbywa się zazwyczaj bez żadnych przeszkód — tak, jakby nic się nie wydarzyło, a operacja wejścia-wyjścia kończy się powodzeniem.

Od tej ostatniej reguły istnieją jednak pewne wyjątki. Jeżeli na przykład jeden z ruterów na drodze transmisji zamiast przestać działać zacznie wysyłać komunikaty o błędzie, o treści takiej jak „host niedostępny”, połączenie zostanie zakończone z rezultatem podobnym do scenariusza z punktu 1. Inna typowa sytuacja zdarza się wtedy, gdy zdalny serwer ma własny system ograniczenia czasu połączenia — wówczas serwer ogranicza czas połączenia i zamyka je, gdy tylko przywrócona zostaje łączność sieciowa.