Programowanie aplikacji sieciowych

Projekt „Programowa i strukturalna reforma systemu kształcenia na Wydziale Mat-Fiz-Inf”.
Projekt współfinansowany ze środków Unii Europejskiej w ramach Europejskiego Funduszu Społecznego.

Człowiek-najlepsza inwestycja

Uniwersytet Marii Curie-Skłodowskiej
Wydział Matematyki, Fizyki i Informatyki
Instytut Informatyki

Programowanie aplikacji sieciowych

Jarosław Bylina
Małgorzata Cudna
Michał Klisowski

UMCS
UNIWERSYTET MARII CURIE-SKŁODOWSKIEJ

Lublin 2012

Instytut Informatyki UMCS
Lublin 2012

Jarosław Bylina
Małgorzata Cudna
Michał Klisowski
PROGRAMOWANIE APLIKACJI SIECIOWYCH

Recenzent: Mateusz Nowak

Opracowanie techniczne: Marcin Denkowski
Projekt okładki: Agnieszka Kuśmierska

Praca współfinansowana ze środków Unii Europejskiej w ramach
Europejskiego Funduszu Społecznego

Publikacja bezpłatna dostępna on-line na stronach
Instytutu Informatyki UMCS: informatyka.umcs.lublin.pl.

Wydawca
Uniwersytet Marii Curie-Skłodowskiej w Lublinie
Instytut Informatyki
pl. Marii Curie-Skłodowskiej 1, 20-031 Lublin
Redaktor serii: prof. dr hab. Paweł Mikołajczak
www: informatyka.umcs.lublin.pl
email: dyrii@hektor.umcs.lublin.pl

Druk
FIGARO Group Sp. z o.o. z siedzibą w Rykach
ul. Warszawska 10
08-500 Ryki
www: www.figaro.pl

ISBN: 978-83-62773-20-6

Spis treści

Wstęp ix

1 Wprowadzenie do programowania na poziomie systemu operacyjnego 1
1.1. Używane środowisko . . . 2
1.2. Dane i wyniki programu . . . 3
1.3. Funkcje systemowe . . . 6
1.4. Pytania i zadania . . . 15

2 TCP/IP 17
2.1. Warstwowa architektura oprogramowania sieciowego . . . 18
2.2. Warstwa sieciowa w Internecie . . . 21
2.3. Warstwa transportowa w Internecie . . . 29
2.4. Pytania i zadania . . . 36

3 DNS, funkcje pomocnicze, kolejność bajtów 39
3.1. Różne użyteczne funkcje . . . 40
3.2. Nazwy domenowe — DNS i resolver . . . 46
3.3. Pytania i zadania . . . 51

4 Gniazda UDP 53
4.1. Krótkie wprowadzenie . . . 54
4.2. Schemat komunikacji . . . 54
4.3. Podstawowe funkcje gniazd UDP . . . 55
4.4. Przykład — usługa echo . . . 59
4.5. Właściwości protokołu — kiedy i jak używać gniazd UDP . . 64
4.6. Pytania i zadania . . . 64

5 Gniazda klienckie TCP 67
5.1. Krótkie wprowadzenie . . . 68
5.2. Schemat komunikacji procesów klienta i serwera TCP . . . 68
5.3. Podstawowe funkcje gniazd klienckich TCP . . . 68
5.4. Przykład — klient usługi czasu dobowego . . . 71

5.5. Pytania i zadania 74

6 Gniazda serwerowe TCP 75
6.1. Funkcje gniazda serwera na przykładzie usługi echo 76
6.2. Rodzaje serwerów TCP 85
6.3. Pytania i zadania 85

7 Opcje gniazd 87
7.1. Sterowanie opcjami gniazd 88
7.2. Przegląd najważniejszych opcji 91
7.3. Pytania i zadania 94

8 Demony i usługi 95
8.1. Procesy w Uniksie 96
8.2. Sygnały 99
8.3. Multipleksacja wejścia/wyjścia 100
8.4. Serwery współbieżne 103
8.5. Demony 105
8.6. Pytania i zadania 108

9 Protokoły warstwy aplikacji 109
9.1. Poczta elektroniczna 110
9.2. WWW 124
9.3. Pytania i zadania 133

10 Projektowanie i implementacja protokołów warstwy aplikacji 137
10.1. Podział danych na części 138
10.2. Reprezentacja danych 139
10.3. Kody odpowiedzi i informacje diagnostyczne 140
10.4. Stany protokołów 140
10.5. Wykorzystanie istniejących rozwiązań 142
10.6. Różne uwagi 143
10.7. Pytania i zadania 143

A Jak to jest w Pythonie? 145
A.1. Interfejs gniazd 146
A.2. Serwery i demony 153
A.3. Moduły wyższego poziomu 157

B Jak to jest w Javie? 165
B.1. Adresy, numery portów a nazwy usług, kolejność bajtów . . . 166
B.2. Gniazda TCP 169

B.3. Gniazda UDP 171
B.4. Serwery współbieżne 173
B.5. Zaawansowane operacje z użyciem `java.nio` 174
B.6. Protokoły warstwy aplikacji 177

C Programy narzędziowe i diagnostyczne **181**
C.1. `ifconfig` 182
C.2. `netstat` 183
C.3. `ping` 183
C.4. `traceroute` 184
C.5. `host`, `nslookup`, `dig` 185
C.6. `tcpdump` 186
C.7. Wireshark 187
C.8. `tcpflow` 188
C.9. `telnet` 188
C.10. `netcat` 188
C.11. Informacje diagnostyczne w aplikacjach użytkowych 188
C.12. `strace -e trace=network` 188

Bibliografia **191**

Skorowidz **195**

Wstęp

Skrypt, który oddajemy w ręce czytelnika powstał dzięki wieloletniemu doświadczeniu autorów w nauczaniu przedmiotów informatycznych na Wydziale Matematyki, Fizyki i Informatyki UMCS, w tym przedmiotów takich, jak „Programowanie aplikacji sieciowych", „Programowanie w środowisku systemu Unix", „Programowanie na poziomie systemu operacyjnego", czy też „Programowanie usług sieciowych". Ma on także za zadanie służyć jako pomoc w nauczaniu wymienionych tutaj (i być może innych) przedmiotów na kierunku „Informatyka". Jednakże, ponieważ prezentuje całkiem dużo różnorakich zagadnień związanych z programowaniem aplikacji działających w sieci, to może być również przydatny dla studentów innych kierunków zainteresowanych tematyką skryptu, a także dla absolwentów, którzy chcieliby uzupełnić wiedzę nabytą na studiach jakiś czas temu.

Przez 'programowanie sieciowe' rozumiemy w niniejszym skrypcie programowanie na poziomie protokołów warstwy transportowej — z użyciem protokołów tej warstwy, przez wykorzystanie interfejsu gniazd TCP/IP (dokładniej: gniazd TCP oraz gniazd UDP). Czytelnicy nie znajdą tutaj elementów programowania webowego czy programowania rozproszonego — te sposoby programowania wykorzystują zwykle narzędzia wyższych poziomów.

Podstawowym celem niniejszego opracowania jest więc zapoznanie Czytelnika z programowaniem aplikacji sieciowych na poziomie gniazd — w tym z projektowaniem i tworzeniem oprogramowania klienckiego i serwerowego TCP oraz UDP.

Zakładamy u Czytelnika chcącego posłużyć się naszym skryptem następujące umiejętności:

- poruszanie się w Uniksowym systemie operacyjnym (jak na przykład Linux), w szczególności w powłoce;
- znajomość organizacji systemu plików w takim systemie;
- umiejętność programowania w języku C (a w przypadku dodatków A oraz B — także w Pythonie i Javie, odpowiednio) oraz kompilowania, testowania i poprawiania odpowiednich programów;
- pewnej umiejętności programowania w języku C na poziomie systemo-

wym (aczkolwiek rozdziały 1 oraz 8 zawierają przegląd niezbędnego minimum);
- znajomość podstawowych zasad działania sieci komputerowych (chociaż rozdział 2 wiele tutaj może jeszcze pomóc).

Książka ta zaczyna od przeglądu pewnych podstawowych elementów programowania systemowego pod Uniksem, niezbędnych w dalszej części pracy (rozdział 1). Rozdziały 2–3 opowiadają o podstawowych elementach budowy sieci w teorii i praktyce. Kolejne rozdziały stanowią przegląd zastosowań i programowania gniazd UDP (rozdział 4) oraz TCP (rozdziały 5–6) oraz przegląd i omówienie ich opcji (rozdział 7). Rozdział 8 wraca do wykorzystania elementów systemu operacyjnego i pokazuje sposób konstrukcji serwerów współbieżnych oraz uruchamiania usług (demonów) w Uniksie. Rozdział 9 omawia przykładowe standardowe protokoły warstwy aplikacji, natomiast rozdział 10 przedstawia zasady ich projektowania. Całość kończą dodatki A–B (omawiające podejście do tematyki książki w Pythonie i w Javie), dodatek C (przegląd programów narzędziowych) oraz bibliografia i skorowidz.

Wszystkich Czytelników zachęcamy także do zajrzenia na stronę www:
`http://kokos.umcs.lublin.pl/ksiazka-pas`
gdzie można (po rejestracji i zalogowaniu się) znaleźć materiały uzupełniające do skryptu, erratę (bo błędów nie unikniemy na pewno) oraz forum dla dociekliwych czytelników. Na tym forum będziemy też omawiali — jeśli zajdzie taka potrzeba — rozwiązania zadań i odpowiedzi na pytania, które będzie można znaleźć na zakończenie każdego rozdziału. Szczególnie dotyczy to zadań oznaczonych gwiazdką (*), czyli trudniejszych.

Książka niniejsza powstała w oparciu o wiele prac (pełna bibliografia znajduje się na stronie 191), ale chyba najważniejsza to [56].

Na koniec chcielibyśmy ogromnie podziękować też naszemu wieloletniemu współpracownikowi i przyjacielowi, który nas większości zagadnień zawartych w tej książce nauczył — Przemkowi Mądzikowi.

Rozdział 1

Wprowadzenie do programowania na poziomie systemu operacyjnego

1.1. Używane środowisko **2**
1.1.1. Architektura Uniksowa jako model systemu operacyjnego 2
1.1.2. Język C 2
1.2. Dane i wyniki programu **3**
1.3. Funkcje systemowe **6**
1.3.1. Dokumentacja 6
1.3.2. Ogólne zasady korzystania z funkcji systemowych 8
1.3.3. Podstawowe operacje plikowe 9
1.4. Pytania i zadania **15**

1.1. Używane środowisko

Ze względu na jednolitość całego skryptu musimy przyjąć pewne środowisko stanowiące punkt odniesienia. Wszystkie informacje, przykłady, kody w głównej części książki (poza dodatkami, które traktują właśnie między innymi o aplikacjach sieciowych w innych środowiskach) odnosić się będą do niego.

1.1.1. Architektura Uniksowa jako model systemu operacyjnego

Jako systemu operacyjnego będziemy używać systemu zgodnego ze standardem POSIX, a więc systemu Uniksowego. Większość (być może wszystkie) przykładów z części głównej skryptu powinna działać na dowolnych systemach zgodnych z Uniksem, aczkolwiek my będziemy tutaj przedstawiać kody przetestowane pod Debianem (wersja 5.0/lenny lub wersja 6.0/squeeze)[1].

Użycie systemu Posiksowego ma wiele zalet:

— przenośność na poziomie kodu języka C;
— uporządkowanie i stabilność w filozofii systemu;
— jednolite podejście do wejścia/wyjścia bez względu na szczegóły techniczne;
— tradycyjne, ustalone i sprawdzone, znane od lat idiomy programistyczne;
— dobrze zaimplementowana współpraca z siecią (wynikająca z równoległego i przeplatającego się rozwoju Uniksa i sieci TCP/IP);
— powszechna dostępność dokumentacji.

Poza tym wszystkim, architektura zgodna ze standardem POSIX może służyć jako pewien wyidealizowany model systemu operacyjnego, w którym przestrzeń jądra od przestrzeni użytkownika jest wyraźnie oddzielona; funkcjonuje też wyraźna separacja uprawnień na wielu innych poziomach.

1.1.2. Język C

Językiem używanym tutaj jest język C, a dokładniej kompilator `gcc` w wersji 4.3.2 ze standardową biblioteką języka C *GNU libc* w wersji 2.7. Jednakże większość (może nawet wszystkie) przykładów będzie bez zmian działać w innych wersjach tych kompilatorów oraz bibliotek, a także z całkiem innymi kompilatorami i bibliotekami.

[1] Oczywiście, Debian to dystrybucja systemu GNU/Linux, a więc formalnie nie jest to Unix — jednakże zgodność Linuksa ze standardem POSIX jest bardzo wysoka, wyższa niż niektórych systemów używających nazwy Unix.

Wybór języka C jest tutaj całkowicie oczywisty. Język C jest mianowicie językiem, w którym napisane jest jądro systemu Linux, a więc cały interfejs funkcji systemowych jest w sposób naturalny dostępny w języku C.

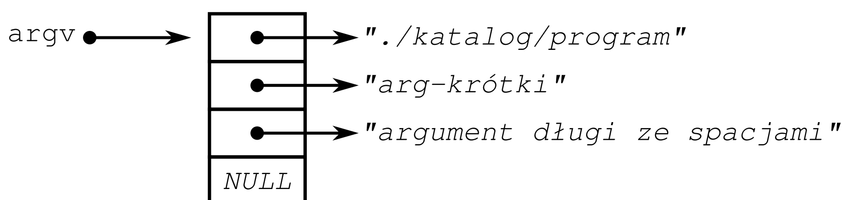
1.2. Dane i wyniki programu

Każdy normalnie uruchomiony program musi mieć kontakt ze swoim środowiskiem, a więc musi mieć możliwość odebrać jakoś dane od otoczenia i jakoś wyniki z powrotem przekazać. Służy temu kilka standardowych dróg.

Argumenty wiersza polecenia. Każdy program napisany w języku C jest realizowany przez wykonanie funkcji głównej o nazwie `main`. Jej prototyp może wystąpić w dwóch poniższych postaciach.

```
int main(void);
int main(int argc, char * argv[]);
```

Pierwsza wersja nagłówka ignoruje kompletnie argumenty, które mogłyby być przekazane programowi przez program nadrzędny (powłokę lub inny program uruchamiający nasz program), ale wersja druga pozwala je odczytać i użyć. W takim przypadku parametr `argc`[2] przechowuje liczbę wszystkich argumentów — wliczając w to samą nazwę ścieżkową programu — a `argv` jest tablicą napisów, przechowujących kolejne argumenty, przy czym `argv[0]` zawiera nazwę ścieżkową programu, natomiast `argv[argc]==NULL` (nie napis pusty!) — patrz także rysunek 1.1.



Rysunek 1.1. Tablica `argv` dla przykładowego wywołania: `./katalog/program arg-krótki 'argument długi ze spacjami'`

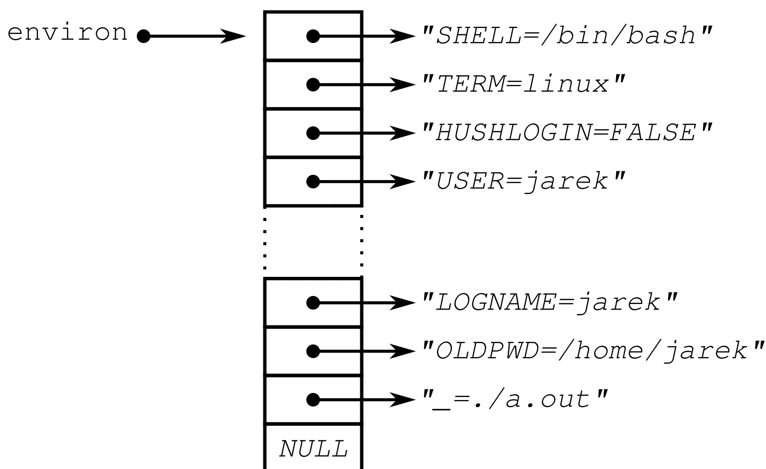
Środowisko wykonania programu Następnymi danymi, które mogą być przekazane programowi przez program uruchamiający, jest tak zwane środowisko wykonania programu, inaczej: zmienne środowiskowe. Jest to zbiór

[2] Nazwy `argc` oraz `argv` są tutaj zwyczajowe i mogą być dowolne inne.

pewnych wartości przypisanych do pewnych identyfikatorów. Można sprawdzić, co jest w środowisku na dwa sposoby. Pierwszy z nich wymaga zadeklarowania zmiennej zewnętrznej (odnoszącej się do zmiennej globalnej ze standardowej biblioteki) w sposób następujący:

```
extern char **environ;
```

Zmienna `environ`[3] jest tablicą napisów, za wyjątkiem jej ostatniego elementu, który jest (analogicznie jak w parametrze `argv`) równy `NULL` (dzięki czemu możemy zorientować się, gdzie kończy się tablica). Każdy z owych napisów jest postaci `"nazwa=wartość"`, przy czym za `wartość` uznawane jest wszystko po pierwszym znaku `'='`. Przykładową zawartość zmiennej `environ` pokazuje rysunek 1.2.



Rysunek 1.2. Przykład tablicy `environ`

Zamiast przeglądać całą tablicę, można także szukać w niej od razu pożądanej nazwy — poprzez użycie funkcji[4]:

```
#include <stdlib.h>
    char *getenv(const char *name);
```

Funkcja ta zwraca wskaźnik do napisu definiującego zmienną środowiskową (oczywiście, spośród elementów zmiennej `environ`) daną przez jej na-

[3] A ta nazwa — w przeciwieństwie do `argc` oraz `argv` — zmieniona być nie może. Dlaczego?

[4] Funkcje języka C będziemy podawali wraz z plikiem (plikami) nagłówkowym, który trzeba dołączyć, by jej użyć — o ile nie będzie to jasne z kontekstu.

zwę (w parametrze `name`). W przypadku braku takowej zwracany jest pusty wskaźnik `NULL`.

Status zakończenia Jak widać na stronie 3, funkcja `main` zwraca wynik typu `int`. Ten wynik, to status (kod) zakończenia programu. Status ten może być odczytany przez program wywołujący dany program i służy zwykle do przekazania informacji o sukcesie lub porażce wykonania.

Każdy program powinniśmy zakończyć wprost — w przeciwnym razie status jego zakończenia może być losowy. Możemy to zrobić kończąc funkcję `main`, jak każdą inną funkcję, za pomocą instrukcji `return` — albo też funkcją systemową `_exit` bądź biblioteczną `exit`, które (w odróżnieniu od instrukcji `return`) zakończą cały program także, gdy są wywołane wewnątrz innej funkcji. Kod zakończenia musimy podać jako argument odpowiedniej instrukcji/funkcji.

Konwencja związana z błędami wykonania jest następująca: kod zakończenia programu równy zero umownie oznacza zakończenie z sukcesem. Przyjmuje się, że w przypadku zakończenia nienormalnego[5] kod ten powinien być różny od zera[6].

Standardowe strumienie wejścia/wyjścia W końcu, każdy normalnie wywołany program ma standardowo otwarte trzy strumienie wejścia/wyjścia (o strumieniach i deskryptorach więcej w podrozdziale 1.3.3), które dostępne są jako obiekty plikowe (typu `FILE *`) zdefiniowane w pliku nagłówkowym `stdio.h` oraz jako deskryptory `0`, `1` i `2`. Wszystkie one są normalnie związane z konsolą, na której program jest wykonywany, ale mogą być przekierowane lub zamknięte. Są to:

— standardowe wejście (`FILE * stdin`, deskryptor `0`) — wszystkie operacje wejścia, które nie wymagają podania pliku (jak `scanf` czy `getchar`), korzystają domyślnie z tego strumienia;
— standardowe wyjście (`FILE * stdout`, deskryptor `1`) — wszystkie operacje wyjścia, które nie wymagają podania pliku (jak `printf` czy `puts`), korzystają domyślnie z tego strumienia;

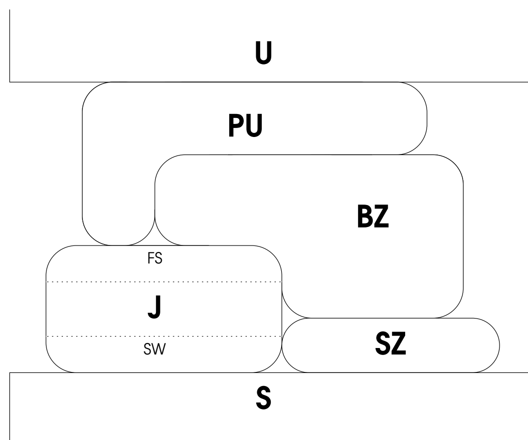
[5] Nie muszą to być błędy wykonywania pewnych funkcji — mogą być inne rodzaje problemów, jak choćby źle wprowadzone dane pochodzące od użytkownika, nieprawidłowy format pliku, nieobsługiwany protokół warstwy aplikacji...

[6] Można używać do tych celów predefiniowanych stałych `EXIT_SUCCESS` oraz `EXIT_FAILURE` — odpowiednio jako kodu zakończenia poprawnego i kodu zakończenia z problemami. Można jednak rozróżniać wiele różnych kodów błędów (co ułatwi późniejszą identyfikację przyczyny). Jednakże, niektóre programy (jak `grep`, `random`, `fsck`) nie stosują się do tej konwencji, zwracając niezerowe wartości także w przypadku sukcesu, wskazując w ten sposób różne rezultaty działania — wtedy oczywiście dokumentacja powinna szczegółowo podawać, co oznaczają poszczególne kody, i które z nich zwracane są w przypadku błędów.

— standardowe wyjście diagnostyczne/błędów (`FILE * stderr`, deskryptor 2) — ten strumień powinien być używany do sygnalizacji błędów i innych informacji diagnostycznych; standardowo używa go na przykład funkcja `perror` służąca do informowania o błędach (patrz podrozdział 1.3.2).

1.3. Funkcje systemowe

Programować będziemy różnego rodzaju usługi sieciowe — na poziomie funkcji systemowych. Co to znaczy? Przyjrzyjmy się schematowi elementów systemu operacyjnego i ich wzajemnych zależności (rysunek 1.3). Nas interesuje najniższy przenośny poziom, to jest poziom wywołań systemowych, a dokładniej interfejsu tych wywołań dostępnego w języku C — czyli właśnie funkcji systemowych.



Rysunek 1.3. System operacyjny jako interfejs między sprzętem a użytkownikiem (U — użytkownik; S — sprzęt; oraz komponenty systemu operacyjnego: J — jądro; SW — sterowniki wewnętrzne; FS — funkcje systemowe; SZ — sterowniki zewnętrzne; BZ — biblioteki zewnętrzne; PU — programy użytkowe)

1.3.1. Dokumentacja

Jak wspomnieliśmy powyżej, ważną zaletą takiego środowiska jest jego dobre udokumentowanie — powszechność i wysoka jakość dokumentacji.

Podstawowym źródłem pomocy jest w Uniksie podręcznik systemowy `man` (skrót od ang. *manual*, podręcznik) zwany tak od nazwy komendy powłoki, która udostępnia jego zawartość. Podręcznik ów jest podzielony na sekcje oznaczane numerami (czasem z dodatkowym przyrostkiem, jak `1M` czy `3C`), przy czym jest to podział tematyczny. Nas będą interesowały głównie

sekcje 2 (funkcje systemowe), 3 (funkcje biblioteczne), 7 (różności), 5 (formaty plików i inne konwencje) — a być może także 1 (programy powłoki) i 4 (specjalne pliki urządzeń).

Każda z sekcji jest natomiast podzielona na strony, których tytułem jest zwykle nazwa opisywanej funkcji, komendy, pliku... Żeby uzyskać dostęp do konkretnej strony podręcznika wywołujemy komendę `man` w poniższy sposób:

```
man sekcja nazwa
```

przy czym `sekcja` może zostać opuszczona, jeśli nie prowadzi to do niejednoznaczności[7].

Tak więc, by znaleźć dokumentację funkcji bibliotecznej `printf` wpisujemy:

```
man 3 printf
```

natomiast, by znaleźć dokumentację komendy powłoki o tej samej nazwie:

```
man 1 printf
```

Warto też wiedzieć, że każda z sekcji ma zwykle specjalną stronę o nazwie `intro` stanowiącą swego rodzaju wstęp do danej sekcji.

Każda strona podręcznika podzielona jest na części, z których bardzo ważną z punktu widzenia używania funkcji systemowych/bibliotecznych (i innych elementów bibliotek) jest ta zatytułowana *SYNOPSIS* (w polskim przekładzie: *SKŁADNIA*). Podaje ona prototyp omawianej (omawianych) funkcji, ponadto pliki nagłówkowe, które należy dołączyć, by móc skompilować bezproblemowo program.

Komenda `man` domyślnie wyświetla stronę w konsoli w sposób pozwalający na jej przewijanie (klawisze strzałek i pokrewne), przeszukiwanie (klawisze `/`, `n` oraz `N`) i powrót do powłoki (klawisz `q`).

Zwyczajowym sposobem zapisywania odsyłacza do odpowiedniej strony podręcznika systemowego jest podanie jej nazwy, po której bezpośrednio (bez odstępów) podana jest w nawiasach okrągłych odpowiednia sekcja — na przykład: `printf(3)`, `printf(1)`, `kill(2)` itp.

Inne sposoby wywołania polecenia `man` pokazują następujące przykłady:

— `man -a printf` — przeglądanie (kolejno) stron o podanej nazwie (tu: `printf`) ze wszystkich sekcji;
— `man -k print` (inaczej: `apropos print`) — wyświetlenie nazw i tytułów (krótkich opisów) stron, które zawierają podany ciąg znaków (tu: `print`);
— `man -f open` (inaczej: `whatis open`) — wyświetlenie tytułów (krótkich opisów) stron o podanej nazwie.

W dzisiejszych czasach wiele serwerów udostępnia dokumentację w formacie `man` poprzez strony www, na przykład:

[7] Jeśli natomiast opuścimy numer sekcji a w kilku sekcjach znajdują się strony o podanej nazwie, wyświetlona zostanie jedna z nich, wybrana w pewnej predefiniowanej kolejności.

`http://matrix.umcs.lublin.pl/cgi-bin/man/man2html`
`http://matrix.umcs.lublin.pl/dwww/man/`

Innym podręcznikiem dostępnym w systemach GNU (a także wielu innych systemach Uniksowych) jest hipertekstowy system dokumentacji `info`. Najpowszechniejszym sposobem dostępu do niej jest przeglądarka konsolowa o tej samej nazwie. Po jej uruchomieniu bez parametrów dostajemy główną stronę dokumentacji zawierającą także spis dostępnych w systemie podstron. Do wyjścia z programu służy także klawisz `q`; do wchodzenia do podstron (są oznaczone znakiem `*`) — klawisze kursora oraz klawisz `Enter`; do poruszania się między stronami — klawisze `p` (strona poprzednia — „w lewo" w drzewie dokumentów), `n` (następna — „w prawo" w drzewie dokumentów), `u` (strona w górę w drzewie dokumentów), `l` (powrót do poprzednio oglądanej).

Wygodniejszy i czytelniejszy interfejs ma opcjonalny (ale również konsolowy) program `pinfo`. Ponadto dokumentacja w tym formacie także bywa udostępniana przez strony www:
`http://matrix.umcs.lublin.pl/cgi-bin/info2www`

1.3.2. Ogólne zasady korzystania z funkcji systemowych

W głównej części skryptu większość omawianych i używanych przez nas funkcji będzie funkcjami systemowymi. Mają one wszystkie pewne wspólne cechy, które tutaj omówimy.

Zwracana wartość. Funkcje systemowe zwracają prawie zawsze liczbę całkowitą ze znakiem. Wartość nieujemna oznacza powodzenie wykonania funkcji i zwykle jest to zero. Inna wartość może wystąpić, jeśli jest taka potrzeba, i jest wtedy wartością znaczącą (jak na przykład w funkcji `open`, która zwraca w przypadku powodzenia deskryptor otwartego pliku, będący właśnie liczbą całkowitą nieujemną).

Sygnalizowanie błędu. Błąd wykonania (lub inna sytuacja wyjątkowa) jest niemal zawsze sygnalizowana przez zwrócenie liczby ujemnej — właściwie zawsze jest to -1. W związku z tym, sama wartość zwrócona nie daje opisu błędu jaki wystąpił. Jest natomiast — w przypadku błędu — jednocześnie ustawiana zmienna globalna[8] `errno` (z pliku nagłówkowego `errno.h`) na liczbę dodatnią będącą kodem błędu.

W pewnych (bardzo nielicznych) przypadkach (na przykład funkcja `getpriority`) wartość ujemna może nie oznaczać błędu, lecz znaczący wynik prawidłowego wykonania funkcji. W takich przypadkach, żeby sprawdzić, czy wywołanie funkcji się powiodło, należy wyzerować wcześniej zmienną `errno`,

[8] Formalnie może być zdefiniowana inaczej, ale zachowuje się jak zmienna globalna.

po czym sprawdzić jej wartość po wykonaniu funkcji — wartość niezerowa oznacza błąd[9].

Przy tej okazji należy wspomnieć jeszcze o sygnalizowaniu błędów użytkownikowi. Najwygodniejsza do tego jest standardowa funkcja `perror` (plik nagłówkowy `stdio.h`), która wyświetla na standardowym wyjściu błędów (patrz też str. 6) standardowy komunikat o błędzie dla bieżącej wartości zmiennej `errno`, wraz z ewentualnym dodatkowym komunikatem (którym tradycyjnie jest nazwa funkcji, która się nie powiodła, całego programu lub też krótki opis okoliczności problemu).

Inne wyniki. Wszelkie inne (niż prosta liczba całkowita) dane, które mogą zostać przez funkcję systemową wydane, są przekazywane przez bufor (bufory), do którego wskaźnik podany został podczas wywołania funkcji. Należy zwykle przed wywołaniem odpowiedni bufor przygotować[10] — można podać adres danej lokalnej lub globalnej, bądź też zarezerwować odpowiednią pamięć przez `malloc`. W tym ostatnim przypadku nie wolno zapomnieć o zwolnieniu pamięci za pomocą `free`, gdy bufor jest już niepotrzebny.

1.3.3. Podstawowe operacje plikowe

W systemach Uniksowych właściwie wszystkie drogi wejścia i wyjścia danych traktowane są jednolicie[11] — podobnie jak zwykłe pliki. Konsekwencją takiego podejścia do rzeczy jest duża jednolitość w operowaniu na tego rodzaju obiektach. W szczególności, wiele funkcji systemowych (jak `read`, `write`, `close`) działa na tego rodzaju obiektach w sposób bardzo podobny — oczywiście w granicach dyktowanych przez ich specyfikę.

Każdy z owych obiektów plikopodobnych jest po otwarciu identyfikowany przez jego *deskryptor* — małą liczbę naturalną (zwykle przydzielana jest pierwsza wolna), która używana jest we wszystkich funkcjach systemowych dla odwołania się do takiego obiektu. Standardowa biblioteka języka C oferuje funkcje wyższego poziomu, a także typ `FILE *`, którego wartości obu-

[9] Ten sposób działa także dla funkcji systemowych zwracających -1 w przypadku błędu. Żadna funkcja systemowa nie zmienia bowiem wartości zmiennej `errno` w przypadku sukcesu, a kody błędów trafiające do `errno` zawsze są niezerowe. Jednakże, funkcje biblioteczne mogą zmieniać wartość tej zmiennej także w przypadku sukcesu, niejako mimochodem — bowiem ich kod składa się zwykle z wywołań wielu funkcji systemowych, z których niektóre mogą się nie powieść.

[10] Inaczej bywa w przypadku niektórych funkcji bibliotecznych, które same to robią, albo też zwracają wskaźnik do pewnych danych statycznych — wszelkie wątpliwości należy wyjaśnić w dokumentacji!

[11] Wyjątkiem są tutaj mechanizmy komunikacji międzyprocesowej pochodzące z Systemu V, ale nimi w niniejszym skrypcie zajmować się nie będziemy.

dowują deskryptory i dostarczają (między innymi) własnego, dodatkowego buforowania[12].

Podstawowe operacje na plikach i obiektach im podobnych realizowane są przez następujące funkcje systemowe:

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
    int open(const char *pathname, int flags);
    int open(const char *pathname, int flags,
             mode_t mode);
#include <unistd.h>
    int close(int fd);
    ssize_t read(int fd, void *buf, size_t count);
    ssize_t write(int fd, const void *buf,
                  size_t count);
#include <sys/types.h>
#include <unistd.h>
    off_t lseek(int fd, off_t offset, int whence);
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
    int stat(const char *path, struct stat *buf);
    int fstat(int fd, struct stat *buf);
#include <sys/stat.h>
    int chmod(const char *path, mode_t mode);
    int fchmod(int fd, mode_t mode);
#include <sys/stat.h>
#include <sys/types.h>
    int mkdir(const char *pathname, mode_t mode);
#include <unistd.h>
    int rmdir(const char *pathname);
    int unlink(const char *pathname);
#include <stdio.h>
    int rename(const char *oldpath,
               const char *newpath);
```

Funkcja `open` odpowiada za otwieranie pliku (i ewentualne tworzenie nowego). Musi być podana jego nazwa ścieżkowa `pathname` oraz różne znaczniki otwarcia `flags`. W przypadku, gdy plik ma być utworzony, należy podać jeszcze jego uprawnienia `mode`.

[12] Tu uwaga: nie należy w stosunku do tego samego obiektu w obrębie jednego programu używać zarówno funkcji systemowych działających na deskryptorach, jak i bibliotecznych używających typu `FILE *` — może to spowodować pomieszanie zapisu lub odczytu ze względu na buforowanie na różnych poziomach.

Wartością zwracaną jest *deskryptor pliku*, który później jest jego identyfikatorem w pozostałych funkcjach systemowych, odnoszących się do otwartego pliku (parametr `fd` powyżej).

Znaczniki `flags` są bitową sumą[13] różnych opcji wyrażonych stałymi, z których niektóre omówione są poniżej (**dokładnie jedna z pierwszych trzech** musi być podana):

— `O_RDONLY` — otwarcie pliku w trybie tylko do odczytu;
— `O_WRONLY` — otwarcie pliku w trybie tylko do zapisu;
— `O_RDWR` — otwarcie pliku w trybie zarówno do zapisu, jak i do odczytu;
— `O_CREAT` — utworzenie nowego pliku (o ile on nie istnieje);
— `O_EXCL` — ma sens tylko z opcją powyższą (`O_CREAT`) i powoduje niepowodzenie, gdy plik już istnieje; innymi słowy, `O_CREAT|O_EXCL` tworzy nowy plik **tylko**, gdy pliku jeszcze nie ma[14];
— `O_TRUNC` — jeśli to możliwe (między innymi, jeśli tryb otwarcia nie jest `O_RDONLY`), to plik jest ucinany do zerowej długości;
— `O_APPEND` — plik jest otwierany do dopisywania, co oznacza automatyczne przesuwanie wskaźnika pliku na koniec przed każdym wywołaniem funkcji `write` zapisującej dane do tego pliku;
— `O_NONBLOCK` lub `O_NDELAY` — włączany jest tryb nieblokujący dla danego pliku, co oznacza, że operacje na nim nie czekają na gotowość do wykonania się, lecz wracają z błędem (domyślnie pliki otwierane są w trybie blokującym).

Podobnie, jak znaczniki `flags`, jako sumę bitową pewnych stałych można wyrazić uprawnienia `mode` do pliku[15]:

— `S_IRUSR` — właściciel ma prawo odczytu;
— `S_IWUSR` — właściciel ma prawo zapisu;
— `S_IXUSR` — właściciel ma prawo uruchamiania;
— `S_IRWXU == S_IRUSR|S_IWUSR|S_IXUSR`;

[13] Należy więc podawać w języku C owe znaczniki w postaci (na przykład): `O_CREAT|O_WRONLY|O_TRUNC`.

[14] Wykorzystywane jest to do tworzenie *plików blokujących*, które zachowują się jak prosty semafor binarny (stany: ‘plik jest’ oraz ‘pliku brak’). Jeśli dwa (lub więcej) procesy chcą użyć pewnego zasobu na wyłączność, mogą spróbować utworzyć plik o ustalonej (tej samej) nazwie ścieżkowej z opcjami `O_CREAT|O_EXCL`. Ponieważ funkcja systemowa `open` jest atomowa (czyli nie może być przerwana przez wykonywanie innych operacji), to uda się ta operacja tylko jednemu; drugi dostanie w jej wyniku błąd. Będzie wiedział więc, że zasób jest zajęty i poczeka na jego zwolnienie (czyli usunięcie pliku przez proces, który zajął zasoby pierwszy). Taki semafor jest oczywiście nieobowiązkowy, w tym sensie, że programy muszą współpracować, żeby plik blokujący pełnił swoją funkcję.

[15] Uprawnienia efektywne do tworzonego pliku są jednak wyliczane jako `mode&~umask`, a więc dodatkowo z użyciem standardowej maski `umask` tworzenia plików, która jest atrybutem każdego procesu (i często jest równa `S_IWGRP|S_IWOTH`) — patrz też strona 106.

— S_IRGRP — grupa ma prawo odczytu;
— S_IWGRP — grupa ma prawo zapisu;
— S_IXGRP — grupa ma prawo uruchamiania;
— S_IRWXG == S_IRGRP|S_IWGRP|S_IXGRP;
— S_IROTH — inni mają prawo odczytu;
— S_IWOTH — inni mają prawo zapisu;
— S_IXOTH — inni mają prawo uruchamiania;
— S_IRWXO == S_IROTH|S_IWOTH|S_IXOTH.

Uprawnienia plików można zmieniać funkcjami chmod (dowolnego pliku, przez podanie jego ścieżki) oraz fchmod (pliku otwartego, przez podanie jego deskryptora).

Plik otwarty należy zamknąć (funkcja close), gdy nie jest już potrzebny[16].

Funkcja lseek przesuwa wskaźnik pliku (czyli miejsce, do którego odnosi się następna operacja czytania lub pisania) o podaną liczbę offset bajów (liczba dodatnia zawsze oznacza przesunięcie w stronę końca, ujemna — w stronę początku) względem miejsca podanego jako whence:

— SEEK_SET — względem początku pliku;
— SEEK_CUR — względem bieżącej pozycji wskaźnika pliku;
— SEEK_END — względem końca pliku.

Do odczytu z pliku służy funkcja read, która dostaje w parametrach deskryptor fd pliku, adres buf pamięci do przechowania danych oraz liczbę count bajtów do odczytania. Zwraca liczbę bajtów realnie odczytanych (o którą też przesuwany jest wskaźnik pliku), która może być mniejsza od count (może to być nawet 0), gdy danych dostępnych (na przykład z powodu końca pliku) jest mniej — lub −1 oznaczające błąd.

Analogiczną funkcją jest write służącą do zapisu danych. Tu oczywiście buf musi wskazywać na dane przygotowane do zapisu.

Plik może zostać usunięty funkcją unlink[17] oraz przeniesiony w inne miejsce struktury katalogowej (lub pod inną nazwę) funkcją rename.

Podstawowe operacje na katalogach realizują funkcje mkdir (utworzenie katalogu, znaczenie mode jak dla open) oraz rmdir (usunięcie pustego katalogu).

[16] Aczkolwiek, deskryptory są zamykane automatycznie przez system, gdy program korzystający z nich się kończy — co ważne jest w przypadku awaryjnego zakończenia procesu.

[17] Tak naprawdę, unlink nie usuwa pliku, lecz usuwa dany wpis o nim w jego katalogu. Plik jest usuwany (to znaczy: jego bloki są zwracane do puli bloków wolnych), gdy nie jest wpisany już w żadnym katalogu **oraz** nie jest otwarty przez żaden proces. Ten sam plik może być wpisany w kilku katalogach (lub w tym samym pod kilkoma nazwami) i nazywa się to *dowiązaniem twardym*, które można wykonać nieomawianą tutaj funkcją link.

W końcu, funkcje `stat` oraz `fstat` zwracają informacje o podanym (za pomocą nazwy ścieżkowej `path` w `stat` lub deskryptora `fd` w `fstat`) pliku. Adres `buf` musi wskazywać tutaj na strukturę `struct stat`, gdzie dane o pliku się znajdą po wywołaniu, a której definicja jest następująca:

```
struct stat {
    dev_t     st_dev;
      /* ID urządzenia zawierającego plik */
    ino_t     st_ino;
      /* numer i-węzła (inode) */
    umode_t   st_mode;
      /* uprawnienia do pliku */
    nlink_t   st_nlink;
      /* liczba dowiązań twardych */
    uid_t     st_uid;
      /* ID właściciela */
    gid_t     st_gid;
      /* ID grupy */
    dev_t     st_rdev;
      /* ID urządzenia (jeśli plik specjalny) */
    off_t     st_size;
      /* całkowity rozmiar w bajtach */
    blksize_t st_blksize;
      /* wielkość bloku dla I/O systemu plików */
    blkcnt_t  st_blocks;
      /* liczba zaalokowanych bloków */
    time_t    st_atime;
      /* czas ostatniego dostępu */
    time_t    st_mtime;
      /* czas ostatniej modyfikacji */
    time_t    st_ctime;
      /* czas ostatniej zmiany */
};
```

Z powyższych pól pewnego komentarza wymaga pole przechowujące uprawnienia (`st_mode`) — oprócz uprawnień (które można normalnie sprawdzać bitowo, na przykład przez `(buf->st_mode)&S_IROTH`) zawiera ono także informacje o typie pliku (jednym z siedmiu standardowo w Uniksie stosowanych). Można go sprawdzić za pomocą następujących makr preprocesora zwracających odpowiedź na podane pytania (w nawiasach podano standardowe jednoznakowe oznaczenia danych typów plików, znane na przykład z polecenia `ls`):

— `S_ISREG(buf->st_mode)` — czy to zwykły plik (`-`)?
— `S_ISDIR(buf->st_mode)` — czy to katalog (`d`)?

— `S_ISCHR(buf->st_mode)` — czy to urządzenie znakowe (`c`)?
— `S_ISBLK(buf->st_mode)` — czy to urządzenie blokowe (`b`)?
— `S_ISFIFO(buf->st_mode)` — czy to łącze nazwane (`p`)?
— `S_ISLNK(buf->st_mode)` — czy to dowiązanie symboliczne (`l`)?
— `S_ISSOCK(buf->st_mode)` — czy to gniazdo (`s`)?

Listing 1.1 pokazuje użycie funkcji systemowych w prostym programie kopiującym plik.

Listing 1.1. Program kopiujący plik

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

#define N 1024

void koniec(char *mess)
{
  perror(mess);
  exit(EXIT_FAILURE);
}

int main(int argc, char *argv[])
{
  char buf[1024];
  int fdsrc, fddst, przeczytane;
  struct stat stat_buf;

  if (argc != 3) {
    fprintf(stderr,
      "Błąd: Wymagane dokładnie dwa argumenty\n");
    exit(EXIT_FAILURE);
  }
  fdsrc = open(argv[1], O_RDONLY);
  if (fdsrc < 0)
    koniec("Błąd open (pierwszy plik)");
  if (fstat(fdsrc, &stat_buf) < 0)
    koniec("Błąd stat (pierwszy plik)");
  fddst =
      open(argv[2],
           O_WRONLY | O_TRUNC | O_CREAT,
           stat_buf.st_mode
             & (S_IRWXU | S_IRWXG | S_IRWXO));
```

```
    if (fddst < 0)
38    koniec("Błąd␣open␣(drugi␣plik)");
    while ((przeczytane = read(fdsrc, buf, N))) {
40    if (przeczytane < 0)
        koniec("Błąd␣read␣(pierwszy␣plik)");
42    if (write(fddst, buf, przeczytane) < 0)
        koniec("Błąd␣write␣(drugi␣plik)");
44  }
    close(fdsrc);
46  close(fddst);
    return EXIT_SUCCESS;
48 }
```

1.4. Pytania i zadania

1. Program z listingu 1.1 zaopatrz w dodatkowe funkcjonalności:
   — możliwość podawania więcej niż dwóch argumentów (wtedy ostatni jest katalogiem, do którego mają być skopiowane pliki podane w pozostałych argumentach);
   — zachowywanie dokładnych uprawnień pliku kopiowanego (w listingu 1.1 uprawnienia te są maskowane przez `umask`);
   — możliwość podawania w pierwszym argumencie katalogu (wtedy drugi też musi być katalogiem istniejącym lub nowym i całe poddrzewo katalogowe jest kopiowane);
   — w przypadku błędów związanych z plikami, wyświetlanie w komunikacie o błędzie nazwy pliku, którego ten błąd dotyczy;
   — jeśli użytkownik poda jako pierwszy argument programu nazwę pliku, który można otworzyć, ale którego nie będzie można skopiować (np. istniejący katalog), program nie powinien tworzyć (ani niszczyć zawartości) pliku zadanego w drugim argumencie;
   — odmowa lub prośba o potwierdzenie nadpisywania plików istniejących.
2. Napisz program, który dla każdego argumentu wiersza poleceń wyświetla jego atrybuty w czytelnej dla człowieka formie (analogicznie do standardowego programu `stat`).
3. * Napisz program otrzymujący w argumencie wiersza poleceń nazwę katalogu. Program wyświetla sumę rozmiarów wszystkich plików znajdujących się w tym katalogu i jego podkatalogach. Pomijane są dowiązania symboliczne. Rozszerz jego funkcjonalność tak, by działał dla dowolnej liczby argumentów (każdy traktując osobno, a także podsumowując ca-

łość) oraz bez argumentów — wtedy wykonuje pracę dla katalogu bieżącego.

4. Napisz program tworzący wszystkie brakujące katalogi w podanej ścieżce dostępu (analogicznie do polecenia powłoki `mkdir` z opcją `-p`). Na przykład poniższe wywołanie powinno utworzyć w bieżącym katalog katalog `kat1`, w nim `kat2`, a w nim `kat3`.

```
./program kat1/kat2/kat3
```

5. * Napisz program usuwający całe poddrzewo katalogowe, którego korzeń podany jest w argumencie (innymi słowy: usuwający niepusty katalog wraz z zawartością).

Rozdział 2

TCP/IP

2.1. Warstwowa architektura oprogramowania sieciowego . . **18**
2.1.1. Model odniesienia OSI 19
2.1.2. Model odniesienia TCP/IP 20
2.2. Warstwa sieciowa w Internecie **21**
2.2.1. Protokół IP 22
2.2.1.1. Format datagramu IP 22
2.2.1.2. Adresy IP 25
2.2.1.3. IPv6 28
2.2.2. Protokół ICMP 28
2.3. Warstwa transportowa w Internecie **29**
2.3.1. Protokół TCP 30
2.3.1.1. Format segmentu TCP 31
2.3.1.2. Fazy połączenia TCP 32
2.3.2. Protokół UDP 35
2.4. Pytania i zadania **36**

2.1. Warstwowa architektura oprogramowania sieciowego

Oprogramowanie sieciowe zorganizowane jest w stos *warstw* w celu zmniejszenia jego złożoności. Niższe warstwy są warstwami bardziej specyficznymi dla danego rodzaju sprzętu czy użytej topologii sieci. Wyższe są bardziej niezależne od konkretnej technologii. Warstwy niższe udostępniają warstwom wyższym określone usługi ukrywając przed nimi niepotrzebne szczegóły. Jest to podejście analogiczne do hermetyzacji (enkapsulacji) w programowaniu obiektowym — klasa udostępnia tylko ściśle określony interfejs i nie trzeba znać szczegółów użytych algorytmów i struktur danych, żeby z niej korzystać.

Takie podejście znacznie ułatwia tworzenie oprogramowania sieciowego. Żeby stworzyć oprogramowanie jednej warstwy, nie jest konieczna dokładna znajomość szczegółów wszystkich pozostałych. Możliwe jest też dokonywanie zmian w poszczególnych warstwach bez wpływu na pozostałe.

Dla przykładu programista tworzący przeglądarkę internetową nie musi znać specyfikacji kart sieciowych zainstalowanych w komputerach, na których przeglądarka będzie działała, topologii sieci lokalnych, w których znajdują się te komputery, czy rodzaju kabli, którymi będą przesyłane dane. Musi tylko znać zestaw usług udostępnianych przez odpowiednią warstwę oprogramowania sieciowego (udostępnianych np. za pomocą interfejsu gniazd opisanego w dalszych rozdziałach). Dzięki temu nie będzie musiał tworzyć osobnej wersji przeglądarki np. dla sieci Ethernet i osobnej dla Wi-Fi. Podobnie twórcy sterowników kart sieciowych nie muszą wiedzieć, jakiego typu aplikacje będą z tego sprzętu korzystać — muszą tylko zadbać o to, żeby ich warstwa oprogramowania udostępniała usługi, z których chcą skorzystać wyższe warstwy.

Każda warstwa porozumiewa się (oczywiście korzystając z usług niższych warstw) z tą samą warstwą na innym komputerze. Sposób takiej komunikacji (zbiór zasad dotyczących kolejności, formatu i znaczenia wymienianych wiadomości) nazywa się *protokołem* danej warstwy.

W celu lepszego zrozumienia idei warstw i komunikacji między nimi możemy posłużyć się poniższym przykładem prezentującym uproszczony model warstwowego oprogramowania sieciowego.

Dwie osoby chcą wymienić jakieś informacje. Decydują się na rozmowę za pomocą komunikatora internetowego. Najwyższą warstwą są tutaj te osoby. Protokołem tej warstwy jest język, w którym rozmawiają. Każda z osób przekazuje informacje drugiej osobie, ale fizycznie przekazanie wiadomości polega na oddaniu jej niższej warstwie — wpisaniu jej w odpowiednie okno komunikatora internetowego. Proces komunikatora musi przekazać tę wiadomość odpowiedniemu procesowi działającemu na komputerze rozmówcy wysyłając mu dane tak sformatowane, żeby ten mógł je odczytać (i np. podzielić na

osobne wiadomości). Komunikator też musi poprosić o przesłanie wiadomości niższą warstwę — zleca wysłanie tej wiadomości systemowi operacyjnemu. Natomiast system operacyjny zleca przesłanie danych sterownikowi karty sieciowej. Po drugiej stronie wszystko odbywa się w odwrotnym kierunku — sterownik karty sieciowej po odebraniu danych przekazuje je systemowi operacyjnemu, system — procesowi komunikatora, a komunikator prezentuje wiadomość drugiemu rozmówcy.

W następnych punktach przedstawimy przykłady dwóch ważnych modeli sieci — *model OSI* i *model TCP/IP*. Modele te określają zestaw warstw i zadania każdej z nich.

2.1.1. Model odniesienia OSI

Model OSI (ang. *Open Systems Interconnection*) jest standardem zaproponowanym przez organizację ISO (ang. *International Standards Organization*). Co prawda protokoły zaproponowane razem z modelem OSI prawie nie są używane, a istniejące architektury nie są nigdy całkowicie z nim zgodne, ale sam model jest często traktowany jako wzorzec, do którego porównuje się inne modele i architektury, a istniejące protokoły zawsze próbuje się przypisywać do jednej z warstw tego modelu. Model ten składa się z siedmiu warstw, z których każda pełni konkretną, dobrze określoną rolę.

Warstwa fizyczna zajmuje się przesyłaniem pojedynczych bitów pomiędzy urządzeniami sieciowymi. Ta warstwa zajmuje się fizycznymi właściwościami łącza takimi jak: wybór napięć, częstotliwość, czas trwania pojedynczego bitu, itp.

Warstwa łącza danych (kanałowa) zajmuje się przesyłaniem danych pomiędzy urządzeniami podłączonymi do wspólnego kanału komunikacyjnego. Jej zadaniem jest też korygowanie błędów mających swoje źródło w warstwie fizycznej oraz sterowanie dostępem do wspólnego nośnika. Podstawową jednostkę informacji w tej warstwie nazywa się zazwyczaj *ramką* (ang. *frame*).

Warstwa sieciowa zajmuje się przesyłaniem danych pomiędzy urządzeniami mogącymi znajdować się w różnych sieciach. Jest to jedyna warstwa, dla której ma znaczenie topologia sieci. Podstawą jednostkę informacji w tej warstwie nazywa się zwykle *datagramem*[1].

[1] Często (może nawet częściej) używa się też nazwy *pakiet* (np. pakiet IP), jednak jest to nazwa ogólniejsza, oznaczająca dowolną jednostkę danych przesyłaną przez sieć komputerową. Natomiast datagramem określa się samodzielną jednostkę zawierającą wszystkie niezbędne informacje do przesłania jej przez sieć bez wcześniejszej wymiany danych [35]. W dokumencie [8] np. używa się nazwy datagram dla pojedynczej jednostki transmisji, a nazwy pakiet dla całego datagramu lub jego fragmentu (jeżeli datagram został poddany fragmentacji).

Warstwa transportowa najczęściej zapewnia wolne od błędów dwupunktowe połączenie gwarantujące dostarczenie danych w kolejności wysyłania (strumień bajtów). W tym przypadku używa się nazwy *segment* dla jednostki przesyłanych danych. Możliwe jest też przesyłanie w tej warstwie osobnych pojedynczych wiadomości (nazywanych też wtedy datagramami) bez nawiązania połączenia oraz bez gwarancji dostarczenia i poprawnej kolejności.

Warstwa sesji zarządza sesjami pomiędzy komunikującymi się programami. Może zajmować się uwierzytelnianiem, punktami kontrolnymi umożliwiającymi wznawianie transmisji, itp.

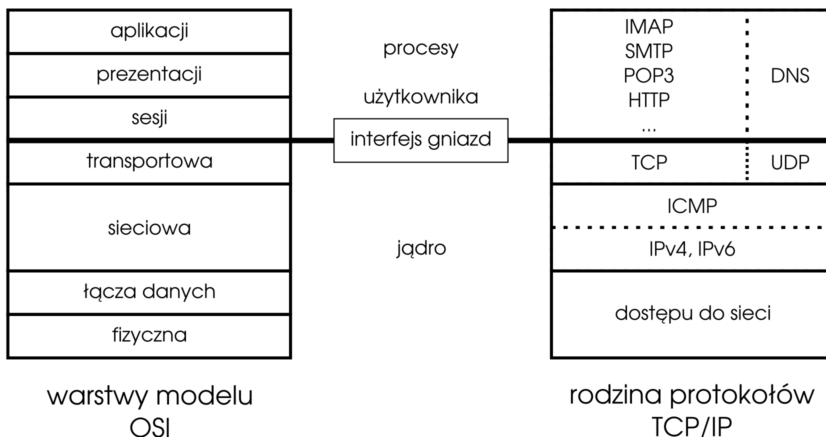
Warstwa prezentacji umożliwia komunikowanie się ze sobą systemów korzystających z różnych reprezentacji danych. Mimo, że różne systemy mogą używać różnego kodowania znaków, kolejności bajtów itp. to warstwa prezentacji dba o to, żeby dane przesyłane były w pewnym wspólnym ustalonym formacie.

Warstwa aplikacji (zastosowań) jest najwyższą warstwą. Warstwa ta, korzystając z usług niższych warstw, przesyła dane otrzymane bezpośrednio od aplikacji.

2.1.2. Model odniesienia TCP/IP

Drugim ważnym modelem jest model odniesienia TCP/IP. W przeciwieństwie do modelu OSI sam teoretyczny model nie jest aż tak przydatny (jest zbyt specyficzny dla protokołów TCP/IP, żeby opisywać jakąkolwiek inną architekturę), natomiast w powszechnym użyciu jest związana z nim rodzina protokołów — jest to rodzina protokołów używanych w Internecie. Często używa się po prostu modelu OSI jako modelu odniesienia, a TCP/IP jako opisu rodziny protokołów sieciowych. Model TCP/IP różni się od modelu OSI przede wszystkim brakiem warstw sesji i prezentacji (warstwa aplikacji przejmuje większość ich zadań) oraz tym, że poniżej warstwy sieciowej określa się tylko jedną warstwę — *warstwę dostępu do sieci* (nazywaną też *warstwą host-sieć*). Zazwyczaj implementacja protokołów warstw poniżej warstwy aplikacji jest dostarczana razem z systemem komputerowym (np. w jądrze systemu operacyjnego lub w postaci biblioteki). Naturalne miejsce na interfejs programistyczny (API) jest więc pomiędzy warstwą aplikacji a transportową. Przykładem takiego właśnie interfejsu jest interfejs gniazd opisany w rozdziałach 4–6. Rysunek 2.1 pokazuje przybliżone odwzorowanie protokołów TCP/IP na model OSI oraz umiejscowienie ich implementacji i interfejsu gniazd w systemach uniksowych.

Nazwy datagram używa się w przypadku usługi zawodnej i bezpołączeniowej (tak jest np. w przypadku warstwy sieciowej w Internecie i protokołu IP). Zawodność oznacza brak gwarancji dostarczenia pakietu lub poinformowania o jego niedostarczeniu.



Rysunek 2.1. Model OSI a protokoły TCP/IP

Dokładne opisy roli i sposobu działania poszczególnych warstw można znaleźć m.in. w książkach [58] i [33].

W dalszej części tego rozdziału omówione zostaną protokoły warstw sieciowej i transportowej z rodziny protokołów TCP/IP. Omówione zostaną przede wszystkim te protokoły i te ich cechy, których zrozumienie jest pomocne lub wręcz niezbędne do poprawnego i efektywnego tworzenia oprogramowania warstwy aplikacji za pomocą interfejsu gniazd — głównego tematu tej książki.

2.2. Warstwa sieciowa w Internecie

Oprogramowanie warstwy sieciowej w Internecie zostało zaprojektowane z myślą o łączeniu ze sobą mniejszych tworzących go sieci. Zadaniem tej warstwy jest udostępnienie warstwie transportowej usługi polegającej na możliwości przesyłania datagramów oraz ukrycie przed tą warstwą szczegółów dotyczących topologii sieci, w których znajdują się komunikujące się ze sobą komputery, oraz topologii ewentualnych sieci pośredniczących w przesyłaniu danych.

Zadanie to realizuje *protokół IP* (ang. *Internet Protocol*). Pomagają mu w tym protokoły takie jak protokół sterujący *ICMP* (ang. *Internet Control Message Protocol*) umożliwiający testowanie i diagnozowanie sieci, czy wykorzystywane przez rutery protokoły wyznaczania tras[2].

Zdecydowanie najczęściej używanym typem komunikacji w Internecie jest *transmisja jednostkowa* (ang. *unicasting*). Polega ona na wysyłaniu danych

[2] Opisane np. w pracy [32].

do dokładnie jednego adresata. Innymi typami są *rozgłaszanie* (ang. *broadcasting*) i *rozsyłanie grupowe* (ang. *multicasting*). Rozgłaszanie polega na wysłaniu danych do wszystkich hostów w zadanej sieci. Rozsyłanie grupowe — na wysłaniu danych do określonej grupy hostów, niekoniecznie znajdujących się w tej samej sieci. Tych typów komunikacji nie będziemy w tej książce dokładniej omawiać.

2.2.1. Protokół IP

Protokół IP jest protokołem odpowiedzialnym za przesyłanie danych w warstwie sieciowej Internetu. Obecnie w odniesieniu do oryginalnego protokołu IP (opisanego w [46]) używa się często nazwy *IPv4* w odróżnieniu od jego nowszej wersji *IPv6* opisanej przez nas dalej w podrozdziale 2.2.1.3.

Protokół IP jest zawodny, tzn. oprogramowanie warstwy sieciowej wysyła pakiet pod zadany adres, ale nie ma żadnej gwarancji, że on tam dotrze. Jest też bezpołączeniowy. Każdy datagram jest obsługiwany niezależnie od pozostałych. Dwa wysłane bezpośrednio po sobie datagramy mogą iść do celu inną drogą i dotrzeć w innej kolejności, niż były wysłane. O niezawodność i synchronizację przesyłanych danych troszczą się wyższe warstwy. W przypadku użycia w warstwie transportowej protokołu TCP dba o to właśnie ten protokół. W przypadku UDP, jeżeli zachodzi taka potrzeba, muszą o to zadbać protokoły warstwy zastosowań.

Oprogramowanie IP hosta konstruuje po prostu na podstawie danych otrzymanych od warstwy transportowej datagram IP, zaopatruje go w nagłówek z odpowiednimi informacjami — w tym odpowiedni docelowy adres IP — i wysyła go przez odpowiedni interfejs sieciowy. Format datagramu IP opisany jest w podrozdziale 2.2.1.1, a format i znaczenie adresów IP w podrozdziale 2.2.1.2.

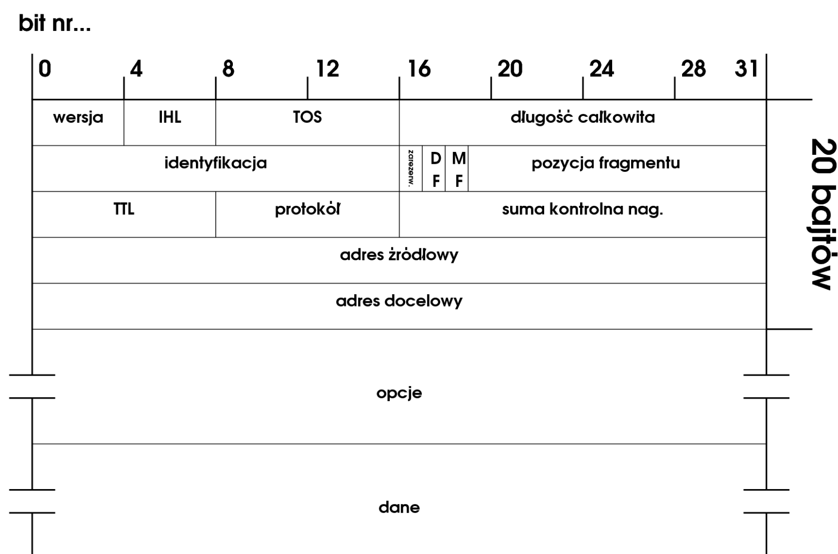
2.2.1.1. Format datagramu IP

Datagram IP składa się z nagłówka i części danych. Nagłówek ma część stałą o długości 20 bajtów i część opcjonalną (jej długość zależy od występujących opcji).

Zazwyczaj większość danych znajdujących się w nagłówku IP nie jest interesująca dla programisty (wyjątkiem są adresy IP — źródłowy i docelowy). Przypisanie odpowiednich wartości tym danym, jak i wysyłanie poszczególnych pakietów IP, odbywa się najczęściej w systemowym oprogramowaniu IP (w jądrze systemu) i jest zwykle całkowicie poza kontrolą programisty. Wartości niektórych pól nagłówka można modyfikować za pomocą odpowiednich opcji gniazd. Opcje te nie dają jednak możliwości odczytania wartości tych pól w przychodzących pakietach. Za pomocą funkcji `getsockopt` można

tylko sprawdzić domyślną wartość tych pól ustawianą w wysyłanych przez system pakietach[3]. Opcje gniazd opisane są w rozdziale 7.

Budowa datagramu IP przedstawiona jest na rysunku 2.2.



Rysunek 2.2. Format datagramu IP

- Czterobitowe pole *wersja* oznacza wersję protokołu. Dla IPv4 jest to po prostu 4.
- Pole *długość nagłówka* (ang. *IHL — Internet Header Length*) określa długość nagłówka w 32-bitowych słowach (maksymalna długość tego 4-bitowego pola to 15 co daje 60 bajtów — maksymalnie 40 bajtów pozostaje na część opcjonalną).
- Ośmiobitowe pole *typ usługi* (ang. *TOS — Type Of Service*) służy do określenia wymagań pakietu dotyczących jego transportu (np. czy ważniejsze jest minimalne opóźnienie czy maksymalna przepustowość bądź niezawodność). Dokładna definicja tego pola zmieniała się wielokrotnie. Historia tych zmian wraz ze wskazaniami na dokumenty RFC opisujące kolejne definicje znajduje się w sekcji 22 dokumentu [51]. Jest to jedno z pól, które może być modyfikowane za pomocą opcji gniazd (opcja `IP_TOS`).
- Pole *długość całkowita* określa długość całego datagramu w bajtach. Pole jest 16-bitowe, co daje maksymalny rozmiar datagramu 65535 bajtów.

[3] Tak naprawdę możemy uzyskać bezpośredni dostęp do wszystkich pól nagłówka IP korzystając z tzw. gniazd surowych (wartość `SOCK_RAW` użyta jako drugi argument funkcji `socket`) i opcji `IP_HDRINCL` wymuszającej podanie przez program nagłówka IP razem z danymi.

Każdy host musi być przygotowany do przyjęcia datagramów o rozmiarze 576 bajtów [46]. W związku z tym wiele protokołów korzystających z UDP w warstwie transportowej (np. DNS) ogranicza się do przesyłania 512 bajtów danych na raz (żeby zmieścić się w 576 bajtach po dodaniu nagłówków UDP i IP). Protokół TCP sam zajmuje się dzieleniem danych użytkownika więc to ograniczenie nie ma tu znaczenia.

- Pole *identyfikacja*, bit *DF* (ang. *Don't Fragment*), bit *MF* (ang. *More Fragments*) oraz pole *pozycja fragmentu* dotyczą fragmentacji datagramów. Fragmentacja, czyli dzielenie datagramu na mniejsze części (fragmenty), umożliwia przesłanie go z wykorzystaniem warstwy kanałowej o wartości MTU[4] mniejszej niż rozmiar tego datagramu. Ustawiony bit DF nie pozwala ruterom na fragmentację datagramu[5].
- Ośmiobitowe pole *czas życia* (ang. *TTL — Time To Live*) (możliwe do ustawienia za pomocą opcji `IP_TTL`) zmniejszane jest o jeden przez każdy ruter, przez który przechodzi pakiet[6]. Jeżeli pole to osiągnie wartość zero przed dotarciem do hosta docelowego, to pakiet jest odrzucany, a do hosta źródłowego wysyłany jest odpowiedni komunikat ICMP (patrz podrozdział 2.2.2). Zapobiega to nieskończonemu krążeniu pakietu w sieci np. na skutek błędnych wpisów w tablicach rutingu.
- Pole *protokół* numer protokołu wyższej warstwy korzystającego z danego datagramu (np. ICMP, TCP, UDP).
- Pole *suma kontrolna nagłówka* weryfikuje poprawność danych znajdujących się w nagłówku przychodzącego pakietu.
- 32-bitowe pola *adres źródłowy* oraz *adres docelowy* zawierają odpowiednio adres hosta wysyłającego i hosta, do którego datagram ma być dostarczony. Więcej o adresach IP w podrozdziale 2.2.1.2.
- Pole *opcje* — to pole jest dość rzadko używane i nie będziemy go tutaj dokładnie opisywać. Dostęp do opcji IP można uzyskać za pomocą opcji gniazd `IP_OPTIONS`. Lista opcji IP dostępna jest pod adresem `http://www.iana.org/assignments/ip-parameters`.

[4] MTU (ang. *Maximum Transmission Unit* — rozmiar największej jednostki danych możliwej do przesłania przez daną warstwę).

[5] Można go ustawić za pomocą opcji `IP_DONTFRAG` w niektórych systemach uniksowych (ale nie w Linuksie) lub za pomocą opcji `IP_DONTFRAGMENT` w systemach z rodziny Windows.

[6] Dokument [46] definiuje to pole jako maksymalny czas życia pakietu w sekundach — ruter przetwarzający pakiet dłużej niż sekundę powinien o odpowiednio więcej zmniejszyć wartość pola. W praktyce rutery zmniejszają zawsze pole o jeden i pole oznacza po prostu maksymalną liczbę przeskoków. Analogiczne pole nagłówka datagramu IPv6 interpretowane jest już właśnie w ten sposób.

2.2.1.2. Adresy IP

Każdy host i ruter w Internecie ma jednoznacznie identyfikujący go adres IP. Adres IP jest 32-bitową liczbą zapisywaną najczęściej jako cztery liczby z zakresu 0–255 oddzielone kropkami, np. `212.182.0.171`. Ściślej mówiąc adres IP identyfikuje nie urządzenie a interfejs sieciowy urządzenia (połączenie z siecią), zatem urządzenia połączone z więcej niż jedną siecią (jak np. rutery) mają więcej adresów IP.

Klasy adresów. Adres IP składa się z części identyfikującej sieć i części identyfikującej hosta w danej sieci. Pierwotnie adresy podzielone były na pięć klas przedstawionych na rysunku 2.3.

klasa A	0	identyfikator sieci (7 bitów)	identyfikator hosta (24 bity)
klasa B	10	identyfikator sieci (14 bitów)	identyfikator hosta (16 bitów)
klasa C	110	identyfikator sieci (21 bitów)	identyfikator hosta (8 bitów)
klasa D	1110	adres rozsyłania grupowego (28 bitów)	
klasa E	11110	zarezerwowane (27 bitów)	

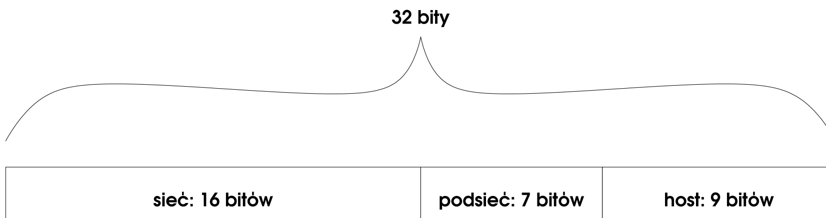
Rysunek 2.3. Podział adresów IP na klasy

W zależności od klasy *identyfikator sieci* i *identyfikator hosta* zajmowały w adresie odpowiednio: 8 i 24 bity dla klasy A (możliwych 16777214 hostów w każdej sieci), 16 i 16 bitów dla klasy B (65534 hosty) oraz 24 i 8 bitów dla klasy C (254 hosty)[7]. Klasa D została przeznaczona na potrzeby rozsyłania grupowego, a adresy klasy E zostały zarezerwowane do przyszłych zastosowań. Numerami sieci zarządza organizacja ICANN (ang. *The Internet Corporation for Assigned Names and Numbers* — Internetowa Korporacja ds. Nadawania Nazw i Numerów).

[7] Liczby możliwych hostów są o 2 mniejsze niż liczba możliwych identyfikatorów, ponieważ adresy złożone z samych zer lub samych jedynek są traktowane w specjalny sposób. Same jedynki w miejscu hosta oznaczają wszystkie hosty w danej sieci (rozgłoszenie). Same zera oznaczają daną sieć (w miejscu identyfikatora sieci) lub danego hosta (identyfikator hosta). Same zera mogą pojawić się w pakiecie IP tylko jako adres źródłowy u hosta, który jeszcze nie zna odpowiednich wartości (podczas ich uzyskiwania).

Podsieci. Mechanizm podsieci polega na tym, że sieć można podzielić na mniejsze sieci (podsieci), wydzielając z pola przeznaczonego na identyfikator hosta część bitów na identyfikator podsieci.

Np. dla sieci klasy B identyfikator hosta (16 bitów) można podzielić na identyfikator podsieci (7 bitów) i identyfikator hosta (9 bitów), jak jest to przedstawione na rysunku 2.4. W obrębie sieci możliwych wtedy jest 128 podsieci po 510 hostów. Granicę pomiędzy identyfikatorem podsieci a identyfikatorem hosta określa się za pomocą 32-bitowej *maski podsieci*. W masce tej wszystkie bity przeznaczone na identyfikator sieci i podsieci ustawione są na 1, a pozostałe na 0. Zapisuje się ją w notacji takiej jak adresy IP (w przedstawionym przykładzie będzie to `255.255.254.0`) lub podaje po prostu długość przedrostka (liczbę jedynek począwszy od lewej strony maski) — tutaj będzie to 23 (16 bitów identyfikatora sieci + 7 bitów identyfikatora podsieci). Adresy sieci lub podsieci przedstawia się często w postaci z długością przedrostka dodaną po `/`, np. `192.168.2.0/23`.



Rysunek 2.4. Podział sieci klasy B na 128 podsieci

Maskę podsieci w danym hoście można sprawdzić m.in. za pomocą programu `ifconfig` (lub `ipconfig` w systemach Windows).

CIDR. Obecnie nie zwraca się uwagi na podział na klasy. Przydzielony identyfikator sieci może być dowolnej długości. Rozwiązanie to nosi nazwę *CIDR* (ang. *Classless InterDomain Routing* — bezklasowy ruting międzydomenowy) i jest przedstawione w dokumencie [25]. Utrudnia ono nieco pracę ruterom, ale pozwala na elastyczniejszy przydział adresów sieciowych (rozmiary sieci mogą być lepiej dopasowane do potrzeb danej organizacji).

NAT. Kolejnym wartym wspomnienia mechanizmem związanym z adresami IP jest *NAT* (translacja adresów sieciowych — ang. *Network Address Translation*) opisany w [54]. Mechanizm ten wykorzystuje pojęcie portu protokołów transportowych (patrz podrozdział 2.3) oraz wydzielenie puli adresów do użytku tylko wewnątrz sieci (*prywatnych adresów* IP) — te adresy nie mogą pojawić się w samym Internecie. Idea NAT opiera się na tym, że liczba dostępnych numerów portów (16-bitowa) jest dużo większa niż liczba

portów rzeczywiście wykorzystywanych nawet po zsumowaniu liczb portów wykorzystywanych przez wszystkie komputery w danej sieci. W każdym pakiecie opuszczającym sieć modyfikowana jest para: adres źródłowy i port protokołu. Adresy prywatne zamieniane są podczas tego przekształcenia na tzw. *adresy publiczne*. W pakietach przychodzących do sieci wykonywane jest przekształcenie odwrotne, ale dotyczące docelowego adresu IP i na jego podstawie pakiet dostarczany jest do odpowiedniego hosta[8]. Zadanie to wykonuje tzw. konwerter NAT (zintegrowany często z ruterem czy firewallem).

Zarezerwowana pula adresów prywatnych [52], to:

```
10.0.0.0       -   10.255.255.255/8
172.16.0.0     -   172.31.255.255/12
192.168.0.0    -   192.168.255.255/16
```

Twórca programów operujących w warstwie aplikacji musi zdawać sobie sprawę z licznych konsekwencji stosowania tego mechanizmu (opisanych w [22]). Oprócz tego, że NAT narusza pewne podstawowe założenia leżące u podstaw architektury TCP/IP (np. założenie, że każdy adres IP jednoznacznie identyfikuje konkretne urządzenie) lub warstwowej architektury oprogramowania sieciowego (założenie niezależności warstw — tutaj warstwa sieciowa musi brać pod uwagę dane warstwy transportowej czyli porty), to niektóre aplikacje nie będą mogły poprawnie funkcjonować w obecności NAT. Będzie tak w przypadku protokołów umieszczających adresy IP lub numery portów wśród przesyłanych danych (np. protokół FTP). Niestety konwerter NAT nie będzie nic wiedział o tych danych. Możliwe jest oczywiście zmuszenie mechanizmu NAT do współpracy z zadaną aplikacją przez zastosowanie odpowiedniego oprogramowania (tzw. bramy warstwy aplikacji — ang. *Application Level Gateway*), ale oznaczać to może konieczność tworzenia nowego oprogramowania NAT dla każdego nowego protokołu. NAT może zepsuć też możliwość wykorzystania *dobrze znanych portów* (patrz podrozdział 2.3) na hostach po stronie „prywatnej" konwertera NAT — tylko jeden host może być odwzorowany z wykorzystaniem zadanego numeru portu.

Pętla zwrotna. Dowolny adres z zakresu `127.0.0.0/8` może być używany do testowania tzw. *pętli zwrotnej*. Pakiety wysyłane pod taki adres pojawiają

[8] Tak naprawdę pojęcie NAT odnosi się ogólnie do mechanizmu zmieniającego informacje o adresach IP w pakietach przechodzących przez urządzenie (np. ruter). Najprostszy typ NAT (Basic NAT, One-to-One NAT) przekształca tylko adresy IP (numerów portów nie) i jest to np. sposób na połączenie dwóch sieci z niekompatybilnym adresowaniem. NAT uwzględniający numery portów nazywa się NAPT (ang. Network Address Port Translation), ale ponieważ jest dużo częstszy, to nazwa NAT zwykle używana jest w odniesieniu właśnie do niego (tak jak w naszym opisie). Terminologia ta opisana jest w dokumencie [55]. Inne spotykane nazwy dla NAPT, to PAT (Port Address Translation), One-to-Many NAT, IP Masquerade, NAT Overload.

się z powrotem jako pakiety wejściowe warstwy IP hosta wysyłającego je. Do dostarczenia tych pakietów nie jest w ogóle używana sieć. Mechanizm ten jest najczęściej używany do testowania programów sieciowych z użyciem jednego komputera i bez wykorzystania rzeczywistej sieci. Zazwyczaj do tego celu używa się adresu `127.0.0.1` zdefiniowanego też jako stała `INADDR_LOOPBACK`.

2.2.1.3. IPv6

Protokół IPv6 jest w zamierzeniu następcą IPv4. Podstawowymi dokumentami opisującymi ten protokół są dokumenty [13] i [26]. Z punktu widzenia programisty warstwy aplikacji niezwykle ważny jest także [21] opisujący rozszerzenia interfejsu gniazd związane z obsługą IPv6.

Obecnie (tzn podczas pisania tej książki) prawie cały ruch sieciowy w dalszym ciągu używa protokołu IPv4. W związku z tym koncentrujemy się głównie na tej wersji protokołu.

Jedną z głównych motywacji wprowadzania nowego protokołu jest wyczerpywanie się puli dostępnych adresów IP. Główną zmianą w nagłówku IPv6 w porównaniu do IPv4 jest więc zmiana długości adresu na 128-bitowy. Poza tym, zmiany w budowie datagramu IP polegają głównie na odchudzeniu nagłówka i zmianie znaczenia i nazw niektórych pól (np. pole *typ usługi* na pole *klasa ruchu*, czy pole *czas życia* na pole *liczba przeskoków*).

Adres IPv6 zapisuje się zwykle za pomocą ośmiu 16-bitowych liczb oddzielonych dwukropkami. Liczby te zapisane są za pomocą czterech cyfr szesnastkowych każda (przy czym początkowe zera można pominąć). Dozwolone jest pominięcie ciągu bloków składających się wyłącznie z zer. Jeżeli po usunięciu zer liczba kolejnych dwukropków następujących po sobie byłaby większa niż dwa, to można zostawić tylko te dwa (ale tylko raz w całym adresie). Przykładowy adres IPv6 może wyglądać tak: `2001:0DB8:0000:0000:0008:0800:200C:417A`, a w skróconej wersji: `2001:DB8::8:800:200C:417A`.

Pewne różnice w pisaniu programów korzystających z protokołu IPv6 w porównaniu do IPv4 będą opisywane przy okazji omawiania kolejnych tematów związanych z programowaniem interfejsu gniazd[9].

2.2.2. Protokół ICMP

Jeszcze jednym ważnym protokołem warstwy sieciowej, o którym powinien wiedzieć twórca aplikacji sieciowych jest protokół *ICMP* (ang. *Internet Control Message Protocol* — internetowy protokół komunikatów sterujących) zdefiniowany w [45]. ICMP wysyła swoje dane korzystając z datagramów IP tak, jak protokoły wyższej warstwy, jednak ze względu na

[9] Jednakże niniejszy skrypt skupia się na użyciu protokołu IPv4.

swoją funkcję przynależy do warstwy sieciowej i jest integralną częścią każdej implementacji protokołu IP. ICMP służy m.in. do testowania sieci oraz zgłaszania błędów związanych z dostarczaniem pakietów IP.

Programy użytkowe bardzo rzadko korzystają bezpośrednio z tego protokołu — często robi to automatycznie oprogramowanie warstwy sieciowej w jądrze i w wielu przypadkach wysłanie lub odebranie komunikatu ICMP jest całkowicie poza kontrolą programisty warstwy aplikacji. Część komunikatów ICMP jest co prawda dostarczana do procesów użytkownika, ale nie da się z nich korzystać za pomocą zwykłych gniazd UDP lub TCP[10].

Błędami zgłaszanymi przez zwykłe gniazda TCP i UDP związanymi z protokołem ICMP są:

— błąd `EHOSTUNREACH` zwrócony (tzn. umieszczony w zmiennej `errno`) przez funkcję `connect` dla gniazda TCP; przyczyną może być jeden z komunikatów typu *Destination Unreachable* (cel nieosiągalny);
— w przypadku gniazd UDP — błąd `EHOSTUNREACH` lub `ECONNREFUSED`[11] zwrócony przez jedną z funkcji wysyłających lub odbierających dane; błąd `ECONNREFUSED` zwracany jest w przypadku braku procesu oczekującego na dane na zadanym porcie (komunikat *Port Unreachable*); błędy te dla protokołu UDP są zgłaszane tylko w przypadku wywołania wcześniej dla danego gniazda funkcji `connect`.

Przykładowymi aplikacjami użytkowymi korzystającymi z ICMP są programy diagnostyczne takie jak `ping` (używający komunikatów typu *Echo Request* — żądanie echa) bądź `traceroute` opisane przez nas w dodatku C.

2.3. Warstwa transportowa w Internecie

Tak, jak na poziomie niższych warstw podczas opisu protokołu rozpatruje się raczej wysłanie pojedynczej porcji danych — mówimy o odbiorcy i o nadawcy — tak w przypadku protokołów wyższych warstw (od transportowej wzwyż) rozważa się zazwyczaj całą sekwencję danych wysyłanych w obie strony. Taka „rozmowa” musi zostać zainicjowana przez jedną ze stron, a druga musi na nią wcześniej biernie oczekiwać. Pierwsza ze stron nazywana jest *klientem*, a druga *serwerem*.

W przypadku protokołów transportowych Internetu (TCP i UDP) klient musi znać adres programu serwera — adres IP komputera, na którym serwer działa i numer *portu*, z którego korzysta serwer. Port jest szesnastobitową liczbą (0–65535).

[10] Trzeba korzystać z gniazd surowych i wartości `IPPROTO_ICMP` użytej jako trzeci argument funkcji `socket` (patrz podrozdział 4.3.1).

[11] Analogiczny błąd `ECONNREFUSED` w przypadku protokołu TCP nie jest spowodowany komunikatem ICMP — jest wynikiem przesłania segmentu RST protokołu TCP.

Parę ⟨adres IP, port⟩ nazywa się *gniazdem*. W przypadku protokołu połączeniowego (TCP) para takich gniazd jednoznacznie identyfikuje połączenie. Pojęcie gniazda zdefiniowane powyżej nie jest dokładnie jednoznaczne z pojęciem gniazda tworzonego za pomocą wywołania `socket` (patrz podrozdział 4.3.1) z interfejsu gniazd. To drugie jest obiektem programowym (a właściwie systemowym, zarządzanym przez jądro). Z tą samą parą ⟨adres IP, port⟩ może być związanych wiele gniazd systemowych (np. w przypadku serwera współbieżnego TCP — różnić się będą analogiczną parą związaną z drugą stroną połączenia).

Każdy proces korzystający z protokołu transportowego musi zarezerwować sobie port odpowiedniego protokołu (porty UDP i TCP są od siebie niezależne). W przypadku serwera port jest zwykle przypisywany do procesu za pomocą wywołania funkcji `bind`. W przypadku klienta wybór konkretnego portu nie ma zazwyczaj znaczenia i jest dokonywany przez system w momencie nawiązania przez tego klienta połączenia (TCP) lub pierwszego wysłania danych (UDP). Zarówno port źródłowy (używany przez proces wysyłający dane) jak i docelowy umieszczany jest w nagłówku odpowiedniego protokołu (TCP lub UDP). Dane przychodzące w warstwie transportowej są przekazywane do odpowiedniego procesu i gniazda na podstawie obu portów i adresów IP.

Serwery protokołów warstwy aplikacji używają często pewnych standardowych ustalonych portów (patrz podrozdział 3.1.2).

W dalszej części rozdziału omówione ostaną dwa podstawowe protokoły używane w warstwie zastosowań TCP/IP — protokół TCP i protokół UDP.

2.3.1. Protokół TCP

Protokół TCP (ang. Transmission Control Protocol — protokół sterowania transmisją) jest podstawowym protokołem warstwy transportowej TCP/IP używanym przez zdecydowaną większość popularnych aplikacji. Jest on opisany w [48]. Jest to protokół *połączeniowy*, co oznacza, że programy chcące wymieniać dane najpierw nawiązują połączenie, następnie wymieniają dane, a na końcu połączenie zwalniają. Połączenie jest dwukierunkowe.

O popularności protokołu TCP decydują przede wszystkim takie jego cechy jak:

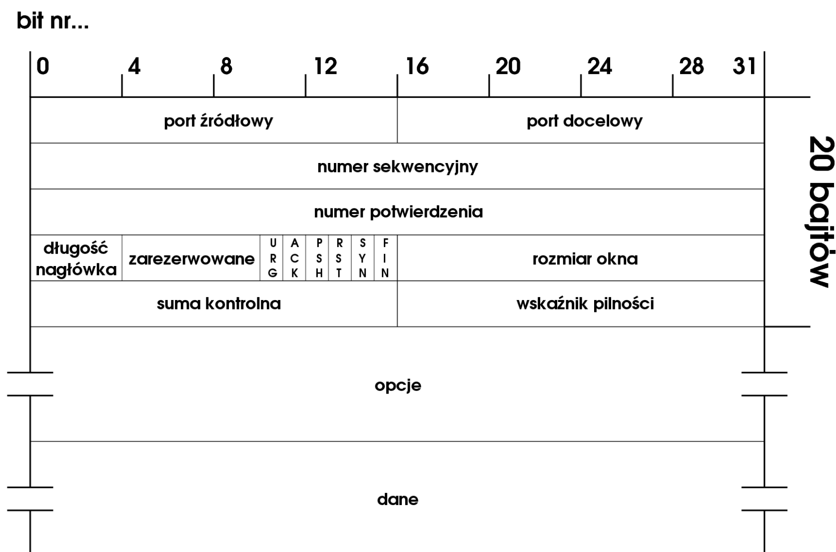
— *niezawodność* — TCP implementuje w sposób niewidoczny dla wyższych warstw mechanizm przesyłania potwierdzeń otrzymanych danych i ewentualnych retransmisji;
— *sterowanie przepływem* (ang. *flow control*) — Oprogramowanie TCP korzysta z mechanizmu *okna przesuwnego* (ang. *sliding window*) — informuje na bieżąco drugą stronę o tym ile danych może przyjąć. Rozmiar okna

(rozmiar danych, które można przyjąć) jest zawsze rozmiarem miejsca w buforze odbiorczym;
— *strumieniowość* — Dane przesyłane w jedną stronę połączenia są traktowane jako strumień bajtów. Wprawdzie powoduje to, że ewentualny podział danych na mniejsze porcje musi być zrealizowany przez aplikację, ale strumieniowość zwalnia protokoły wyższych warstw od konieczności ewentualnego porządkowania kolejności otrzymanych danych — jest to realizowane już przez oprogramowanie TCP.

2.3.1.1. Format segmentu TCP

Najmniejszą jednostką danych rozpatrywaną przez protokół TCP jest segment. Segment składa się z nagłówka i danych. Podobnie jak w przypadku protokołu IP większość pól nie jest bezpośrednio używana przez programistę warstwy aplikacji. Programista nie zajmuje się nawet pojedynczymi segmentami — interesuje go zazwyczaj tylko fakt nawiązania połączenia, przesyłane dane oraz fakt zakończenia połączenia. Tworzeniem odpowiednich segmentów i przesyłaniem ich zajmuje się oprogramowanie TCP — zazwyczaj w jądrze systemu.

Budowa nagłówka TCP przedstawiona jest na rysunku 2.5.



Rysunek 2.5. Format nagłówka TCP

— Pola *port źródłowy* i *port docelowy* zostały omówione na początku bieżącego rozdziału.

— Pola *numer sekwencyjny* i *numer potwierdzenia* oraz flaga *ACK* są potrzebne do implementacji niezawodności — potwierdzeń i retransmisji.
— Ustawienie flagi *URG* oznacza, że pole *wskaźnik pilności* przekazuje informację o położeniu *danych pozapasmowych* (ang. *out-of-band data*, inaczej *pilnych danych* — ang. *urgent data*)[12].
— Flaga *PSH* (ang. *push*) oznacza dane do wysłania i przekazania aplikacji docelowej bez buforowania.
— Flaga *RST* (ang. *reset*) sygnalizuje błąd i oznacza konieczność natychmiastowego zakończenia połączenia. Segment z tą flagą jest wysyłany np. w przypadku otrzymania segmentu, który nie pasuje do żadnego istniejącego połączenia lub w przypadku próby nawiązania połączenia z użyciem portu, który nie jest przypisany do żadnego serwera.
— Flagi *SYN* i *FIN* są używane odpowiednio do nawiązywania i kończenia połączenia — o tym będzie dalej.
— Pole *rozmiar okna* związane jest ze sterowaniem przepływem. Oznacza rozmiar oferowanego okna.
— Przy konstruowaniu *sumy kontrolnej* oprócz nagłówka TCP brane pod uwagę są także oba adresy IP — źródłowy i docelowy, wersja TCP (6) i długość segmentu.
— *opcje* pozawalają na przekazanie niestandardowych informacji, jak np. możliwość interpretowania wielkości okna w jednostkach większych niż bajty[13], czy informacja o konieczności ponownego przesłania pojedynczego fragmentu i równoczesnego potwierdzenia pozostałych (nawet późniejszych).

Segmenty mające ustawione flagi np. RST, SYN, FIN czy ACK nazywa się po prostu odpowiednio segmentem RST, segmentem SYN, segmentem FIN bądź segmentem ACK.

2.3.1.2. Fazy połączenia TCP

Poniżej przedstawione zostaną kolejne fazy połączenia TCP — otwieranie, przesyłanie danych i zamykanie. Podczas tych faz połączenie przechodzi przez kolejne stany zdefiniowane w sekcji 3.2 dokumentu [48][14]. Zdarzeniem

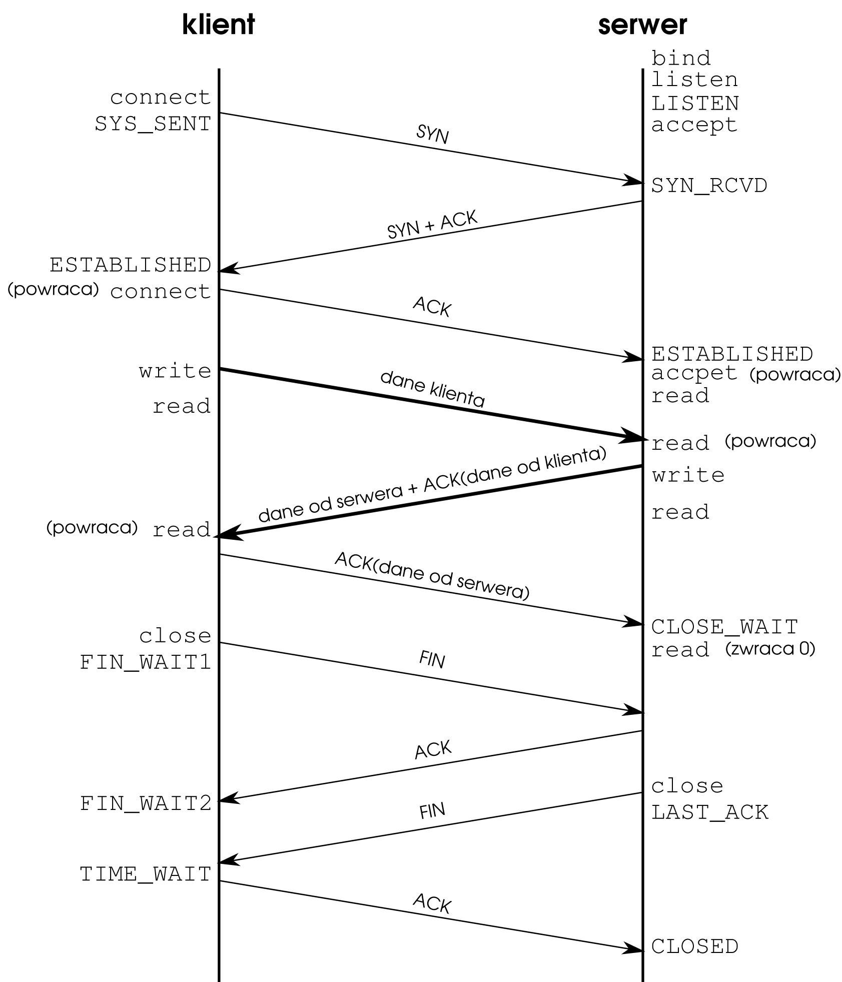
[12] Jest to bardzo rzadko stosowany mechanizm. W rzeczywistości nie pozwala on na wysłanie danych poza kolejnością, a jedynie umożliwia ich obsługę po stronie odbiorcy wcześniejszą, niż obsługa pozostałych danych z tego samego segmentu (a w niektórych przypadkach — wcześniejsze poinformowanie o tym, że takie dane się pojawią). Stosowanie go za pomocą interfejsu gniazd jest możliwe za pomocą flagi `MSG_OOB` użytej podczas wysyłania danych za pomocą funkcji `send` lub `sendto` (patrz podrozdział 4.3.3).

[13] Pole 16-bitowe ogranicza rozmiar okna do 64 bajtów, co jest wartością zbyt małą szczególnie dla połączeń o dużej przepustowości i dużym opóźnieniu.

[14] Przydatnym narzędziem informującym o tym w jakim stanie są istniejące na komputerze połączenia, jest program `netstat` opisany w dodatku C.

powodującym zmianę stanu może być wywołanie funkcji gniazdowej, nadejście odpowiedniego segmentu bądź upłynięcie zadanego czasu.

Poniżej przedstawione są typowe sekwencje przesyłania segmentów związane z kolejnymi fazami. Mogą one ulec zmianie np. w przypadku zagubienia któregoś segmentu i konieczności jego retransmisji. Funkcje, których nazwy są tutaj używane są przedstawione w rozdziałach 4–6, a kolejne stany typowego połączenia TCP przedstawione są na rysunku 2.6.



Rysunek 2.6. Wymiana segmentów i stany połączenia TCP

Nawiązywanie połączenia. Serwer po wywołaniu funkcji `listen` znajduje się w stanie `LISTEN`. W celu nawiązania połączenia klient wysyła segment SYN (wywołując funkcję `connect`). Przechodzi wtedy w stan `SYN_SENT`

w oczekiwaniu na potwierdzenie wysłanego segmentu (odpowiedni segment ACK) i przesłanie segmentu SYN serwera (zaakceptowanie połączenia — serwer musi wcześniej wywołać funkcję `accept`). Serwer odsyła zazwyczaj segmenty SYN i ACK połączone i przechodzi w stan `SYN_RECEIVED` (także: `SYN_RCVD`) w oczekiwaniu na potwierdzenie swojego segmentu SYN. Nadejście odpowiednich potwierdzeń powoduje powrót z funkcji `connect` i `accept` oraz przejście obu procesów do stanu `ESTABLISHED` i wymiany regularnych danych.

Przesyłanie danych. Podczas przesyłania danych oba programy (klient i serwer) niczym się nie różnią z punktu widzenia TCP. Ewentualne różnice narzucone są przez wyższe warstwy. Oba programy mają teraz możliwość wysyłania (np. funkcją `write`) i odbierania (np. funkcją `read`) danych. Wszystkie dane są potwierdzane odpowiedzią z ustawioną flagą ACK W zwykłym blokującym[15] trybie działania funkcja `read` powraca, gdy tylko może odebrać dane, natomiast funkcja `write` powraca od razu po zapisaniu danych do bufora wysyłającego TCP.

Mechanizm potwierdzania wszystkich danych może powodować spory narzut na ilość transmitowanych danych zwłaszcza w przypadku przesyłania ich małymi porcjami. Przesłanie 1 bajtu danych może np. spowodować konieczność przesłania 41-bajtowego datagramu IP (20-bajtów zajmuje nagłówek IP, 20-bajtów — nagłówek TCP i jeden bajt — dane) oraz 40-bajtowego potwierdzenia. TCP stara się na różne sposoby ograniczyć ten narzut:

— Kiedy tylko się da potwierdzenia przesyłane są w segmencie razem z danymi.
— Stosowany jest *algorytm Nagle'a* — jeżeli jakieś wysłane dane nie zostały jeszcze potwierdzone, to TCP kolejne otrzymane dane buforuje do czasu otrzymania potwierdzenia starych danych. Buforowane dane mogą być dzięki temu wysłane w jednym segmencie.
— Stosowany jest *algorytm opóźnionego potwierdzenia* — po otrzymaniu danych, jeżeli nie ma nic do wysłania, TCP zwleka jakiś czas w nadziei na otrzymanie kolejnych danych i wysłanie pojedynczego potwierdzenia[16].

Zwalnianie połączenia. Kolejna faza następuje, gdy któraś ze stron skończy przesyłać dane i prześle segment FIN (zamknięcie aktywne). Zazwyczaj

[15] W trybie nieblokującym wszystkie funkcje powracają od razu bez względu na to, czy udało im się coś zrobić (patrz rozdział 7).

[16] Połączenie tego algorytmu z algorytmem Nagle'a może powodować znaczne opóźnienia. Dzieje się tak gdy program wysyła dane w małych porcjach, a druga strona w żaden sposób na te dane nie odpowiada. Strona wysyłająca oczekuje na szybkie potwierdzenie, a strona odbierająca nie ma danych, z którymi mogłaby je wysłać. W celu rozwiązania tego problemu interfejs gniazd udostępnia opcję `TCP_NODELAY` umożliwiającą wyłączenie stosowania algorytmu Nagle'a.

jest to spowodowane wywołaniem funkcji `close` lub `shutdown` z parametrem `SHUT_WR`[17]. Strona ta oczekuje teraz w stanie `FIN_WAIT_1` na potwierdzenie, a następnie przechodzi w stan `FIN_WAIT_2` w oczekiwaniu na segment FIN od drugiej strony. Po otrzymaniu go przechodzi w stan `TIME_WAIT`. Druga strona (zamknięcie bierne) po otrzymaniu segmentu FIN przechodzi w stan `CLOSE_WAIT`, wysyła potwierdzenie, swój segment FIN (podobnie — `close` lub `shutdown`) i przechodzi w stan `LAST_ACK` — oczekiwanie na ostatnie potwierdzenie wysłanego segmentu FIN.

Odebranie segmentu FIN oznacza, że program nie otrzyma już żadnych danych, ale cały czas może te dane wysyłać. Jest sygnalizowane przez powrót z funkcji `read` z wartością 0. Kończenie połączenia jest właściwie dwoma niezależnymi zakończeniami przesyłania danych w obie strony.

Warto zwrócić uwagę na stan `TIME_WAIT`. Jest to stan, w którym strona aktywnie zamykająca połączenie pozostaje dość długo. Dokument [8] zaleca, aby były to 4 minuty. Jednak w różnych implementacjach TCP czas ten jest różny. Stan ten pozwala na poprawne zakończenie połączenia nawet w przypadku zagubienia ostatniego segmentu ACK. W takim przypadku strona bierna ponownie wyśle segment FIN, a strona aktywna ponowi wysłanie segmentu ACK. Bez utrzymywania jeszcze przez jakiś czas informacji o połączeniu strona aktywna musiałaby odesłać segment RST, co zostałoby zinterpretowane przez drugą stronę jako błąd. Poza tym stan `TIME_WAIT` pozwala na zniknięcie z sieci ewentualnych zagubionych duplikatów[18] segmentów należących do tego połączenia. W przypadku użycia ponownie tej samej pary gniazdowej stare duplikaty mogłyby być potraktowane jako segmenty nowego połączenia.

2.3.2. Protokół UDP

Drugim najważniejszym protokołem warstwy transportowej TCP/IP jest UDP opisany w dokumencie [49]. Protokół UDP jest *datagramowym* protokołem *bezpołączeniowym* pozbawionym takich zalet TCP jak niezawodność, sterowanie przepływem, czy automatyczne porządkowanie przychodzących danych. Oznacza to, że przesyłane dane mogą nie dotrzeć do celu, dotrzeć wielokrotnie lub w innej kolejności niż były wysłane bez żadnej informacji o wystąpieniu błędów.

Jest on jednak wykorzystywany tam, gdzie przynajmniej część z tych cech nie jest potrzebna. Zaletą UDP jest wtedy szybkość działania i prostota implementacji — niepotrzebne jest np. nawiązywanie połączenia, przesyłanie

[17] Ale może być też spowodowane zakończeniem procesu.

[18] Takie duplikaty mogą się pojawić np. w razie zagubienia lub dużego opóźnienia w przesyłaniu segmentów potwierdzających. Druga strona myśląc, że segment wymagający potwierdzenia nie dotarł, wyśle go ponownie.

potwierdzeń, kończenie połączenia czy przechowywanie danych potrzebnych do ewentualnych retransmisji zagubionych pakietów. Jeżeli któraś z wymienionych cech jest potrzebna, to musi być zaimplementowana w warstwie aplikacji.

Protokół datagramowy oznacza, że w przeciwieństwie do protokołu strumieniowego podział danych na porcje jest utrzymywany przez protokół. Jeżeli dane dotrą do celu, to w takich samych częściach jak zostały wysłane. Nie jest możliwe połączenie dwóch datagramów w jeden czy podzielenie datagramu na mniejsze.

Protokół UDP jest w istocie protokołem IP z dodanym mechanizmem portów umożliwiającym przekazanie datagramu do odpowiedniego procesu na hoście docelowym.

Typowe zastosowania UDP to np. przesyłanie danych multimedialnych w czasie rzeczywistym (zgubionych pakietów nie ma sensu retransmitować bo będą nieaktualne) czy proste systemy klient-serwer z krótką wiadomością i krótką odpowiedzią. W przypadku braku odpowiedzi klient po prostu ponowi zapytanie lub wyśle je gdzie indziej. Typowy przebieg komunikacji wymaga wtedy wysłania tylko dwóch datagramów.

Z UDP korzystają też programy wymagające rozsyłania grupowego lub rozgłaszania — te mechanizmy nie są dostępne w TCP.

2.4. Pytania i zadania

Niektóre z zadań mogą wymagać uruchomienia programu z prawami administratora.

1. Napisz program umożliwiający użytkownikowi ustawienie wartości pól TTL i TOS w wysyłanym pojedynczym datagramie UDP. Sprawdź (np. za pomocą programu `tcpdump`) wartości tych pól po dotarciu do hosta docelowego.
2. Po wykonaniu zadań z kolejnych rozdziałów, spróbuj w wybranych programach umożliwić ustawienie (np. za pomocą parametru wiersza poleceń) pola TTL w wysyłanych pakietach IP. Sprawdź jaki wpływ ma ustawienie zbyt małej (uniemożliwiającej pakietom IP dotarcie do celu) wartości pola TTL na działanie protokołu TCP, a jaki na UDP.
3. Prześledź np. za pomocą programu `tcpdump` lub `Wireshark` pakiety przesyłane podczas typowej komunikacji za pomocą protokołów TCP i UDP. Co się dzieje w przypadku próby nawiązania połączenia TCP z portem nieużywanym przez żaden serwer? Jakie pakiety są odbierane od hosta docelowego przy próbie wysłania datagramu UDP pod port niezwiązany z żadnym procesem? Czy interfejs gniazd zwraca wtedy jakieś informacje o błędach?

4. Przetestuj działanie algorytmu Nagle'a. Zrób to za pomocą programu wysyłającego sporo pojedynczych krótkich wiadomości i np. programu `tcpdump`. Zaobserwuj jaka jest różnica w liczbie przesyłanych segmentów po wyłączeniu stosowania algorytmu Nagle'a. Jakie są różnice w działaniu algorytmu Nagle'a pomiędzy sieciami o małym a sieciami o dużym opóźnieniu?
5. Zaobserwuj np. za pomocą programu `netstat` stany TCP, w jakich może znajdować się program. Przez jaki czas po zakończeniu działania programu i wykonaniu zamknięcia aktywnego gniazdo jest jeszcze w stanie `TIME_WAIT`?

Rozdział 3

DNS, funkcje pomocnicze, kolejność bajtów

3.1. Różne użyteczne funkcje **40**
3.1.1. Kolejność bajtów 40
3.1.2. Usługi a porty 41
3.1.3. Gniazdowe struktury adresowe i funkcje przekształcające adresy 43
3.1.3.1. Gniazdowe struktury adresowe 43
3.1.3.2. Funkcje przekształcające adresy . . . 44
3.1.4. Nazwa lokalnego hosta 46
3.2. Nazwy domenowe — DNS i resolver **46**
3.2.1. DNS 46
3.2.2. Resolver 47
3.2.2.1. Funkcje biblioteczne 47
3.3. Pytania i zadania **51**

W tym rozdziale przedstawimy krótko system *DNS* (system nazw domenowych — ang. *Domain Name System*) zapewniający odwzorowanie domenowych nazw komputerów (czytelnych i łatwych do zapamiętania dla człowieka) na adresy IP (są używane przez warstwę sieciową Internetu). Przedstawione też zostaną funkcje z biblioteki języka C realizujące to zadanie z wykorzystaniem DNS oraz lokalnych plików konfiguracyjnych. Zestaw takich funkcji nazywa się *resolverem*. Przedtem jednak przedstawione zostaną inne użyteczne funkcje pomocnicze przydatne przy programowaniu interfejsu gniazd, ale niezwiązane bezpośrednio z przesyłaniem danych. Są to funkcje odpowiadające za przetwarzanie danych związanych z siecią — zmianę kolejności bajtów, odwzorowanie nazw usług (i protokołów warstwy aplikacji) na numery portów czy konwersję pomiędzy różnymi sposobami reprezentacji adresów sieciowych.

3.1. Różne użyteczne funkcje

3.1.1. Kolejność bajtów

Liczby zapisywane za pomocą więcej niż jednego bajtu mogą być przechowywane w pamięci na dwa sposoby:

— od najmniej do najbardziej znaczącego bajtu (ang. *little-endian*, patrz też [57]),
— od najbardziej do najmniej znaczącego bajtu (ang. *big-endian*).

Różne systemy komputerowe używają różnej kolejności bajtów. Np. architektura x86 używana w komputerach PC korzysta z kolejności little-endian. Kolejność bajtów używaną w danym systemie nazywamy *systemową kolejnością bajtów* (ang. *host byte order*). Kolejność bajtów używaną przez protokoły sieciowe nazywamy *sieciową kolejnością bajtów* (ang. *network byte order*). W protokołach TCP/IP używana jest kolejność z najbardziej znaczącym bajtem pierwszym. Funkcjami dokonującymi przekształceń między sieciową i systemową kolejnością bajtów są `htonl`, `htons`, `ntohl` i `ntohs` zaprezentowane na listingu 3.1. W ich nazwach `n` oznacza sieć (ang. *net*), `h` oznacza hosta, `s` — typ `short int`, a `l` — typ `long int`. Co prawda typy te mogą zajmować różną liczbę bajtów w różnych systemach, ale tutaj litera `s` oznacza zawsze wartość 16-bitową (jak np. numer portu TCP lub UDP), a `l` — 32-bitową (jak np. adres IPv4).

Listing 3.1. Funkcje zmieniające kolejność bajtów

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

3 uint32_t htonl(uint32_t hostlong);
```

```
  /* Przekształca 32-bitową wartość hostlong
5    z systemowego na sieciowy porządek bajtów. */

7 uint16_t htons(uint16_t hostshort);
  /* Przekształca 16-bitową wartość hostshort
9    z systemowego na sieciowy porządek bajtów. */

11 uint32_t ntohl(uint32_t netlong);
  /* Przekształca 32-bitową wartość netlong
13   z sieciowego na systemowy porządek bajtów. */

15 uint16_t ntohs(uint16_t netshort);
  /* Przekształca 16-bitową wartość netshort
17   z sieciowego na systemowy porządek bajtów. */
```

3.1.2. Usługi a porty

Porty protokołów TCP i UDP (patrz 2.3) o numerach mniejszych niż 1024 zarezerwowane są na potrzeby standardowych usług i w większości systemów mogą być otwierane jedynie przez procesy działające z prawami administratora. Dzięki temu klient łącząc się z takim portem może założyć, że łączy się ze standardowym programem udostępniającym daną usługę, a nie z przypadkowym programem uruchomionym przez jakiegoś użytkownika danego komputera. Porty takie określane są nazwą *ogólnie znanych* lub *dobrze znanych* (ang. *well-known ports*). Lista dobrze znanych portów wraz z usługami z nimi powiązanymi zarządzana jest przez organizację IANA (ang. *Internet Assigned Numbers Authority* i jest dostępna pod adresem `http://www.iana.org/assignments/port-numbers`. Lokalnie w systemach uniksowych lista taka zawarta jest zazwyczaj w pliku `/etc/services`[1]. Listy te zawierają także tzw. *porty zarejestrowane* z zakresu 1024–49151. W większości systemów porty z tego zakresu mogą być używane przez zwykłych użytkowników. Pozostałe porty (49152–65535) także mogą zwykle być używane przez zwykłych użytkowników, ale są to równocześnie porty przydzielane przez jądro automatycznie połączeniom „wychodzącym".

Przykładowe wiersze z pliku `/etc/services` mogą wyglądać tak:

```
finger    79/tcp
daytime   13/udp
ssh       22/tcp                        # SSH Remote Login Protocol
```

[1] W systemach z rodziny Windows NT pliki konfiguracyjne związane z siecią analogiczne do uniksowych plików takich jak `/etc/services`, `/etc/hosts`, znajdują się w katalogu `%SystemRoot%\system32\drivers\etc\`, gdzie zmienna `%SystemRoot%` określa położenie katalogu systemowego (np. `C:\Windows\`).

```
domain      53/tcp                  # name-domain server
domain      53/udp
www         80/tcp     http         # WorldWideWeb HTTP
pop3        110/tcp    pop-3        # POP version 3
nfs         2049/tcp                # Network File System
nfs         2049/udp                # Network File System
```

Wzajemne odwzorowanie pomiędzy nazwami usług warstwy aplikacji (takimi jak np. http, smtp, finger) i numerami portów realizowane jest za pomocą funkcji `getservbyname` i `getservbyport` przedstawionych na listingu 3.2.

Listing 3.2. Funkcje odczytujące usługi i porty

```
1 #include <netdb.h>

3 struct servent *getservbyname(const char *name,
                                const char *proto);
5
  struct servent *getservbyport(int port,
7                               const char *proto);
```

Funkcje te dla zadanej nazwy usługi (takiej jak np. `"finger"`, `"www"`, czy `"pop3"`) lub zadanego portu zwracają statycznie zaalokowaną strukturę `servent` zdefiniowaną na listingu 3.3:

Listing 3.3. Struktura `servent`

```
1 struct servent {
    char *s_name;       /* nazwa usługi */
3   char **s_aliases;   /* lista aliasów */
    int s_port;         /* numer portu */
5   char *s_proto;      /* protokół */
  }
```

Argument `proto` oznacza wybrany protokół (`"tcp"` lub `"udp"`). Jeżeli będzie to `NULL`, to pasuje dowolny protokół. W razie wystąpienia błędu (np. nie znaleziono usługi lub dana usługa nie jest dostępna dla danego protokołu) funkcje te zwracają `NULL`.

Listing 3.4 przedstawia funkcję zwracającą numer portu w systemowej kolejności bajtów skojarzony z zadaną usługą dostępną za pomocą protokołu UDP.

Listing 3.4. Zamiana nazwy usługi na numer portu

```
  int serv_name2port(char *nazwa_uslugi)
2 {
```

```
    struct servent *serv =
4       getservbyname(nazwa_uslugi, "udp");
    if (serv == NULL) {
6     /* nie znaleziono zadanej usługi */
      return 0;
8   }
    return ntohs(serv->s_port);
10 }
```

3.1.3. Gniazdowe struktury adresowe i funkcje przekształcające adresy

3.1.3.1. Gniazdowe struktury adresowe

Interfejs gniazd może służyć do tworzenia programów komunikujących się w różny sposób — za pomocą protokołu IPv4, IPv6, a także do programowania mechanizmów komunikacji pomiędzy procesami na tym samym hoście (bez użycia protokołów sieciowych)[2]. W każdym z tych typów komunikacji proces dostępny jest za pomocą innego rodzaju adresu. W przypadku protokołów IPv4 i IPv6 jest to adres IP — odpowiednio w wersji 4 lub 6 — wraz z numerem portu, a w przypadku komunikacji międzyprocesowej — nazwa ścieżkowa w obrębie używanego systemu plików. Wszystkie te mechanizmy korzystają z tych samych funkcji wymagających podania adresu (np. `bind`, czy `connect`). Funkcje te (wraz z interfejsem gniazd) powstały wcześniej niż standard ANSI C i wskaźnik `void*` mogący wskazywać na dane różnych typów. W związku z tym w pliku nagłówkowym `sys/socket.h` została zdefiniowana *ogólna struktura adresowa* (listing 3.5).

Listing 3.5. Ogólna struktura adresowa

```
  struct sockaddr {
2   sa_family_t sa_family;
    char sa_data[14];
4 }
```

Jedynym zastosowaniem tej struktury jest rzutowanie wskaźników do konkretnych typów adresowych na typ wskaźnika do tej właśnie struktury.

Dla protokołów IPv4 i IPv6 w pliku `netinet/in.h` zostały zdefiniowane struktury przedstawione na listingu 3.6.

Listing 3.6. Struktury adresowe IP

[2] Są to tzw. gniazda w dziedzinie Unix. Nie będziemy się nimi zajmować w tej książce.

```
struct sockaddr_in {
  sa_family_t sin_family;     /* AF_INET */
  in_port_t sin_port;         /* port */
  struct in_addr sin_addr;    /* adres IP */
};

struct in_addr {
  uint32_t s_addr;            /* adres IP */
};

struct sockaddr_in6 {
  sa_family_t sin6_family;    /* AF_INET6 */
  in_port_t sin6_port;        /* port */
  uint32_t sin6_flowinfo; /* etykieta przepływu */
  struct in6_addr sin6_addr; /* adres IPv6 */
  uint32_t sin6_scope_id;     /* zasięg */
};

struct in6_addr {
  unsigned char s6_addr[16]; /* adres IPv6 */
};
```

Wszystkie pola liczbowe tych struktur używają sieciowej kolejności bajtów.

3.1.3.2. Funkcje przekształcające adresy

Do przekształcania adresów IP między postacią napisów takich jak pokazane w podrozdziałach 2.2.1.2 i 2.2.1.3, a postacią wymaganą przez pola `sin_addr` i `sin6_addr` struktur adresowych służą funkcje przedstawione na listingu 3.7 i zadeklarowane w pliku nagłówkowym `arpa/inet.h`.

Listing 3.7. Struktura `addrinfo`

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

in_addr_t inet_addr(const char *strptr);

char *inet_ntoa(struct in_addr inaddr);

int inet_pton(int family, const char *strptr,
              void *addrptr);

const char *inet_ntop(int family,
                      const void *addrptr,
                      char *strptr, socklen_t len);
```

Funkcje `inet_aton` i `inet_pton` służą do przekształcania napisów (argument `strptr`) na postać liczbową (`addrptr`), a `inet_ntoa` i `inet_ntop` — na odwrót. Funkcje `inet_aton` i `inet_ntoa` obsługują tylko adresy IPv4, a `inet_pton` i `inet_ntop` umożliwiają także obsługę IPv6. Protokół określa się za pomocą parametru `family` — `AF_INET` dla IPv4 i `AF_INET6` dla IPv6.

Funkcja `inet_addr` wykonuje to samo zadanie co `inet_aton`, ale jej użycie jest odradzane. Jako wartość zwraca ona bowiem stałą `INADDR_NONE` w przypadku błędu. Problemem jest to, że stała ta zdefiniowana jako same jedynki (dwójkowo) jest poprawnym rozgłoszeniowym adresem IPv4 — `255.255.255.255`[3].

Dokładny opis parametrów i zwracanych wartości opisanych funkcji można znaleźć w podręczniku systemowym[4].

Listing 3.8 pokazuje przykład użycia funkcji `inet_pton` i funkcji `htons` do wypełnienia gniazdowej struktury adresowej na podstawie danych pobranych z wiersza poleceń.

Listing 3.8. Użycie funkcji `inet_pton`

```
1  int main(int argc, char *argv[])
   {
3    ...

5    struct sockaddr_in addr;
     int ret;
7    if ((ret =
          inet_pton(AF_INET, argv[1],
9                   &addr.sin_addr)) == -1) {
       fprintf(stderr,
11              "Nieobsługiwany protokół\n");
       exit(EXIT_FAILURE);
13   } else if (ret == 0) {
       fprintf(stderr, "Zły format adresu\n");
15     exit(EXIT_FAILURE);
     }
17   addr.sin_family = AF_INET;
     addr.sin_port = htons(atoi(argv[2]));
19
     ...
21 }
```

[3] Taką samą wartość ma zresztą również stała `INADDR_BROADCAST` używana do rozgłaszania.

[4] Przydać się mogą jeszcze stałe `INET_ADDRSTRLEN` i `INET6_ADDRSTRLEN` zdefiniowane w pliku `netinet/in.h` określające maksymalny rozmiar buforów potrzebnych do zapisania w postaci napisu adresów IPv4 i IPv6.

3.1.4. Nazwa lokalnego hosta

Przydatnymi funkcjami umożliwiającymi uzyskanie nazwy lokalnego hosta są `uname` (umożliwia także uzyskanie informacji o systemie operacyjnym i rodzaju sprzętu) oraz `gethostname`. Dokładny opis tych funkcji można znaleźć w podręczniku systemowym.

3.2. Nazwy domenowe — DNS i resolver

Programy sieciowe korzystające z TCP/IP potrzebują do przesłania danych adresu IP odbiorcy. Niestety dla użytkowników tych programów korzystanie z adresów IP wiąże się z pewnymi problemami. Po pierwsze są one trudne do zapamiętania. Po drugie adres IP, pod którym dostępna jest dana interesująca użytkownika usługa, ulegnie zmianie np. po przeniesieniu jej na inny komputer lub po przeniesieniu komputera do innej sieci. Użytkownik potrzebuje adresu mnemonicznego (np. `matrix.umcs.lublin.pl` zamiast `212.182.0.171`) łatwiejszego do zapamiętania niż adresy numeryczne i rzadziej ulegającego zmianie. Obecność dwóch rodzajów adresów wymaga istnienia mechanizmu umożliwiającego konwersję między nimi. Takim mechanizmem jest DNS umożliwiający zamianę adresów IP na *adresy domenowe*.

3.2.1. DNS

DNS jest rozproszonym systemem komputerowym. Działanie tego systemu umożliwiają serwery DNS wraz z protokołem komunikacyjnym umożliwiającym korzystanie z nich. DNS jest opisany w dokumentach [37] i [38].

DNS organizuje nazwy domenowe w hierarchiczną strukturę. Nazwa domenowa składa się z *etykiet* oddzielonych kropkami. Np. domena `www.example.org`[5] zawarta jest w domenie `example.org`, a ta z kolei w domenie najwyższego poziomu (ang. *top-level domain*) `org`. Nazwa złożona z etykiet wszystkich poziomów nazywa się *pełną nazwą domenową* (FQDN — ang. Fully Qualified Domain Name). Np. `www.example.org` jest pełną nazwą domenową, a `www` — nie.

Dane przechowywane w systemie DNS składają się z *rekordów zasobów* (ang. *resource records*). Każdy z nich ma pole *typ* określające znaczenie i format zasobów, których dotyczy. Typy najbardziej nas interesujące, to:

A — odwzorowanie nazwy domenowej na adres IPv4;
AAAA — analogiczny do A, ale dotyczy IPv6;

[5] Domena `example.org` jest jedną z domen zarezerwowanych przez [15] do użytku w dokumentacji lub do celów testowych. Inną często używaną domeną opisaną tam jest `localhost` tradycyjnie skojarzona z adresem pętli zwrotnej (patrz podrozdział 2.2.1.2).

PTR — tzw. *rekord wskaźnikowy*; służy do odwzorowania adresu IPv4 na nazwę domenową; adres IP musi być zapisany w specjalnej postaci — bajty zapisane są w odwrotnej kolejności, a na końcu dołączony jest napis `.in-addr.arpa`, np. dla `212.182.0.171` będzie to `171.0.182.212.in-addr.arpa`;

MX — określenie hosta odpowiedzialnego za pocztę w danej domenie.

Przydatnymi narzędziami służącymi do testowania DNS są programy takie jak `host`, `nslookup` czy `dig` opisane przez nas w dodatku C.

3.2.2. Resolver

Programy użytkowe potrzebują mechanizmu umożliwiającego im w prosty sposób korzystać z systemu DNS. Mechanizm ten nazywany jest *resolverem*[6]. Resolver jest zwykle zaimplementowany jako zestaw funkcji bibliotecznych. Jego podstawowym zadaniem jest zamiana nazwy domenowej na adres IP lub na odwrót. Żeby to osiągnąć resolver nie zawsze musi korzystać z serwerów DNS. Czasem odpowiednie dane może znaleźć zapisane w lokalnych plikach lub cache'u[7].

3.2.2.1. Funkcje biblioteczne

Tradycyjnymi funkcjami realizującymi zadania resolvera są `gethostbyname` i `gethostbyaddr` zadeklarowane w pliku `netdb.h` i przedstawione na listingu 3.9.

Listing 3.9. Funkcje `gethostbyname` i `gethostbyaddr`

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

3 struct hostent *gethostbyaddr(const void *addr,
                                int len, int type);
```

Z funkcjami tymi związana jest globalna zmienna `h_errno` typu `int`, w której umieszczany jest numer błędu w razie jego wystąpienia.

[6] Mechanizm ten w polskiej literaturze nosi różne nazwy, np. *mechanizm odwzorowania adresów* [56] lub *proces określający nazwy* [58]. My będziemy tutaj jednak używać angielskiej nazwy resolver, która jest po prostu prostsza i czytelniejsza.

[7] Dokładne zachowanie resolvera w systemach uniksowych jest zazwyczaj kontrolowane przez kilka plików konfiguracyjnych jak np. `/etc/nsswitch.conf`, `/etc/resolv.conf` czy `/etc/hosts`. Ogólne wskazówki dotyczące implementacji resolvera znajdują się w dokumencie [7] w sekcji 6.1.

Pierwsza z tych funkcji przyporządkowuje nazwom domenowym adresy IP[8], a druga na odwrót. Obie te funkcje zwracają wskaźnik do statycznie zaalokowanej struktury `hostent`[9] zdefiniowanej na listingu 3.10.

Listing 3.10. Struktura `hostent` `inet_pton`

```
struct hostent {
  char *h_name;       /* oficjalna nazwa komputera */
  char **h_aliases;   /* lista aliasów */
  int h_addrtype;     /* typ adresu komputera */
  int h_length;       /* długość adresu */
  char **h_addr_list;/* lista adresów */
}
#define h_addr  h_addr_list[0]
```

W celu obsługi protokołu IPv6 wprowadzono funkcję `gethostbyname2`[10] (funkcja `gethostbyaddr` od początku miała argument `type` umożliwiający zadanie konkretnego typu adresu).

Dokładne informacje o parametrach, zwracanych wartościach i kodach błędów wszystkich wymienionych funkcji można znaleźć w podręczniku systemowym.

Powyższe funkcje są proste w użyciu i często używane, jednak zalecanym i w pełni wspierającym programowanie niezależne od wybranego protokołu sposobem na konwersję pomiędzy nazwami domenowymi i adresami IPv4 są funkcje `getaddrinfo` i `getnameinfo`. Pierwsza z nich przyporządkowuje nazwom domenowym adresy IP, a druga na odwrót.

Funkcja `getaddrinfo` zadeklarowana jest w pliku `netdb.h` (listing 3.11).

Listing 3.11. Funkcja `getaddrinfo`

```
int getaddrinfo(const char *node,
                const char *service,
                const struct addrinfo *hints,
                struct addrinfo **res);
```

Dla zadanych argumentów `node` (adres hosta) i `service` (nazwa usługi lub numer portu — patrz podrozdział 3.1.2) funkcja zwraca w parametrze `res` listę dynamicznie zaalokowanych struktur `addrinfo` (listing 3.12).

[8] Aczkolwiek, w przypadku podania adresu IP jako argumentu, także działa, kopiując ów adres do odpowiednich pól struktury wynikowej.

[9] Funkcje te w związku ze zwracaniem statycznej zmiennej nie są wielowejściowe. W bibliotece glibc (GNU C Library) zostały zdefiniowane analogiczne wielowejściowe funkcje `gethostbyname_r` i `gethostbyaddr_r`.

[10] I analogiczną wielowejściową funkcję `gethostbyname2_r`.

Listing 3.12. Struktura `addrinfo`

```
struct addrinfo {
  int ai_flags;
  int ai_family;
  int ai_socktype;
  int ai_protocol;
  size_t ai_addrlen;
  struct sockaddr *ai_addr;
  char *ai_canonname;
  struct addrinfo *ai_next;
};
```

Aby zwolnić pamięć przeznaczoną na tę listę należy wywołać funkcję `freeaddrinfo`.

Szczegóły związane z tymi funkcjami można znaleźć w podręczniku systemowym. Wyczerpujące omówienie sposobów użycia funkcji `getaddrinfo` można znaleźć w rozdziale 11 książki [56].

Poniżej zaprezentowane są dwie wersje programu wyświetlającego adresy IPv4 hosta zadanego argumentem wiersza poleceń. Program z listingu 3.13 korzysta z funkcji `gethostbyname`, a program z listingu 3.14 — z funkcji `getaddrinfo`.

Listing 3.13. Użycie funkcji `gethostbyname`

```
#include <stdio.h>
#include <stdlib.h>
#include <netdb.h>
#include <sys/socket.h>
#include <arpa/inet.h>

int main(int argc, char *argv[])
{
  char straddr[INET_ADDRSTRLEN];
  struct hostent *he_ptr;
  char **addrptr;

  if (argc < 2) {
    fprintf(stderr, "Brak argumentu\n");
    exit(EXIT_FAILURE);
  }

  he_ptr = gethostbyname(argv[1]);
  if (he_ptr != NULL) {
    addrptr = he_ptr->h_addr_list;
    while (*addrptr) {
      printf("%s\n",
```

```
                inet_ntop(AF_INET, *addrptr, straddr,
24                        INET_ADDRSTRLEN));
          addrptr++;
26      }
      } else {
28      fprintf(stderr, "Błąd!\n");
        exit(EXIT_FAILURE);
30    }
      return 0;
32  }
```

Listing 3.14. Użycie funkcji `getaddrinfo`

```
    #include <stdio.h>
 2  #include <string.h>
    #include <stdlib.h>
 4  #include <netdb.h>
    #include <arpa/inet.h>
 6
    int main(int argc, char *argv[])
 8  {
      char straddr[INET_ADDRSTRLEN];
10    struct addrinfo hints;

12    if (argc < 2) {
        fprintf(stderr, "Brak␣argument\n");
14      exit(EXIT_FAILURE);
      }
16
      memset(&hints, 0, sizeof(hints));
18    hints.ai_family = AF_INET;
      hints.ai_socktype = SOCK_STREAM;
20
      struct addrinfo *ai_ptr = NULL;
22    int errcode;
      if ((errcode =
24         getaddrinfo(argv[1], NULL, &hints,
                       &ai_ptr)) == 0) {
26      while (ai_ptr) {
          printf("%s\n",
28               inet_ntop(AF_INET,
                        &((struct sockaddr_in *)
30                        ai_ptr->ai_addr)->sin_addr,
                        straddr, INET_ADDRSTRLEN));
32        ai_ptr = ai_ptr->ai_next;
        }
```

```
34    } else {
        fprintf(stderr, "Błąd!\n");
36    }
      freeaddrinfo(ai_ptr);
38    return 0;
    }
```

3.3. Pytania i zadania

1. Napisz program wyświetlający informację o używanej w systemie kolejności bajtów.
2. Napisz program mogący przyjmować w argumencie wiersza poleceń nazwę usługi (np. `www`, `smtp`, `rpc`) lub numer portu. Jeżeli podana została nazwa usługi, to program wyświetla odpowiadający jej numer portu. Jeżeli podany został numer portu wyświetlana jest nazwa odpowiedniej usługi.
3. Napisz program sortujący adresy IPv4 otrzymane w argumentach wiersza poleceń. Adresy są sortowane według ich wartości liczbowej (a nie np. leksykograficznie). Np poniższy ciąg jest odpowiednio posortowany:

   ```
   4.3.2.1
   15.10.5.0
   87.246.247.240
   127.0.0.1
   212.182.0.171
   255.255.255.255
   ```

4. Napisz program, który dla hosta zadanego w argumencie wiersza poleceń wyświetli wszystkie jego adresy IP i nazwy domenowe. Host może być zadany zarówno za pomocą adresu IP jak i za pomocą adresu domenowego.
5. Napisz serwer UDP, który po odebraniu pakietu wyświetla na standardowym wyjściu dostępne informacje o nadawcy — jego adresy IP i domenowe.

Rozdział 4

Gniazda UDP

4.1. Krótkie wprowadzenie **54**
4.2. Schemat komunikacji **54**
4.3. Podstawowe funkcje gniazd UDP **55**
4.3.1. Funkcja `socket` 56
4.3.2. Funkcja `bind` 56
4.3.3. Funkcje `recvfrom` i `sendto` 57
4.3.4. Uwaga o funkcji `connect` 58
4.4. Przykład — usługa echo **59**
4.4.1. Program serwera 59
4.4.2. Program klienta 61
4.5. Właściwości protokołu — kiedy i jak używać gniazd UDP **64**
4.6. Pytania i zadania **64**

4.1. Krótkie wprowadzenie

User Datagram Protocol, w skrócie UDP, a w dosłownym tłumaczeniu *Protokół Datagramów Użytkownika*, jest jednym z protokołów sieciowych warstwy transportowej umożliwiających komunikację w sieci komputerowej. Omówienie właściwości tego protokołu Czytelnik znajdzie w rozdziale 2.

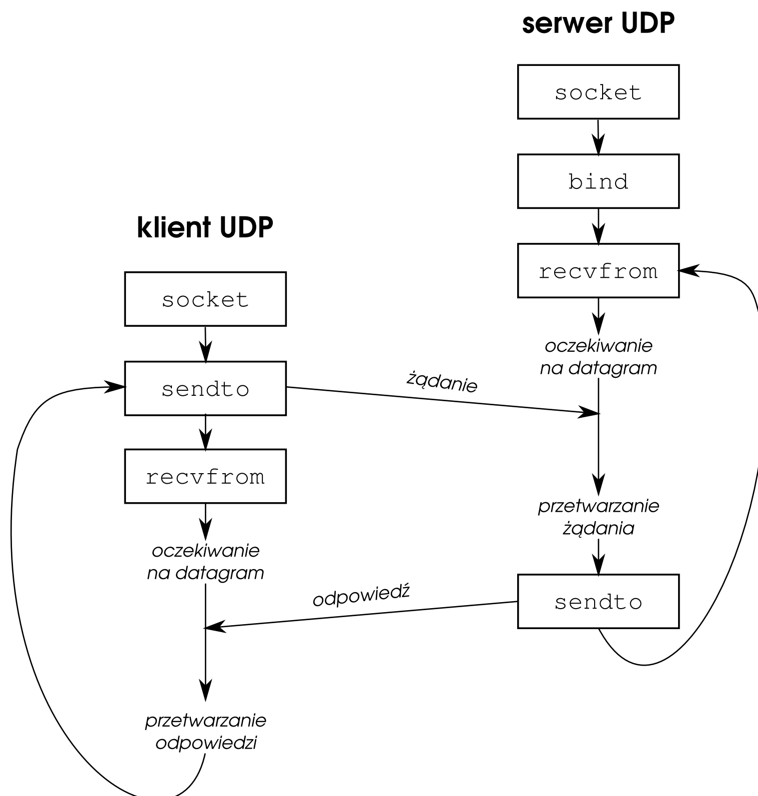
Przy użyciu gniazd UDP dane przesyłane są w postaci pakietów, tutaj nazywanych *datagramami*. Każdy datagram przesyłany jest niezależnie od pozostałych, a między komputerami nie jest nawiązywane połączenie. UDP jest zatem prostym protokołem umożliwiającym szybką transmisję danych, ale przez to zawodnym — istnieje możliwość utracenia datagramów, bądź przesłania ich w innej kolejności. Obsługa wymienionych wyżej błędów możliwa jest w programie użytkowym, lecz należy pamiętać, że nie jest realizowana na poziomie protokołu. Dzięki temu transport danych jest szybszy, bez nakładów poświęcanych na potwierdzanie odbioru, obsługę czasu oczekiwania czy też ponawianie transmisji. Gniazda UDP nie utrzymują połączenia — to samo gniazdo można wykorzystać do komunikacji z wieloma różnymi klientami czy serwerami.

Najbardziej popularnymi zastosowaniami UDP są: system nazw domenowych DNS, sieciowy system plików NFS, prosty protokół zarządzania siecią SNMP, stosowany w transmisji dźwięków mowy *Voice over IP*, *IP tunneling* używany w wirtualnych sieciach prywatnych, a także oprogramowanie gier online. UDP stosowany jest więc w aplikacjach implementujących komunikację typu żądanie-odpowiedź, ale również, jeśli utrata pakietu jest mniej istotna niż generowane przez jego powtórne przesłanie opóźnienie, czyli w aplikacjach mających działać w czasie rzeczywistym.

4.2. Schemat komunikacji

Z dwóch procesów komunikujących się w sieci komputerowej jeden nazywamy serwerem, a drugi klientem. Serwer rozpoczyna pracę jako pierwszy i oczekuje na zgłoszenie klienta. W niektórych przypadkach może to być jedyna różnica między programami, jednak najczęściej serwer udostępnia w sieci pewną usługę, a klient wysyła do niego żądanie i odbiera odpowiedź, na ogół wielokrotnie.

Na rysunku 4.1 przedstawiony został schemat typowej wymiany komunikatów w protokole UDP. Po utworzeniu gniazda funkcją `socket`, serwer dowiązuje do niego swój ogólnie znany port przy pomocy funkcji `bind` i oczekuje na zgłoszenia. W przypadku gniazd datagramowych klient nie ustanawia połączenia z serwerem, tylko bezpośrednio wysyła żądanie. Komunikacja między procesami polega więc jedynie na przesyłaniu datagramów za



Rysunek 4.1. Schemat typowej wymiany komunikatów UDP

pomocą funkcji `sendto` i odbieraniu ich poprzez wywołanie funkcji `recvfrom`. Szczegółowe omówienie użytych funkcji znajdzie Czytelnik w kolejnym podrozdziale.

Serwery pracujące na gniazdach UDP są najczęściej jednowątkowymi serwerami iteracyjnymi. Oznacza to, że jeden proces serwera obsługuje wszystkich klientów, a obsługa klienta realizowana jest bezpośrednio po odebraniu żądania za pomocą jednego i tego samego gniazda. Datagramy, które zostały przesłane, a jeszcze nie zostały pobrane przez program serwera (np. od innych klientów) przechowywane są w niejawnej kolejce do gniazda, a jej przepełnienie skutkuje odrzuceniem pakietu.

4.3. Podstawowe funkcje gniazd UDP

W tym rozdziale omówimy podstawowe funkcje gniazd, służące do ich tworzenia i usuwania, nadawania adresu protokołowego oraz wykonywania operacji wejścia/wyjścia. Szczegóły wywołania dotyczyć będą gniazd data-

gramowych, można jednakże ten rozdział traktować jako ogólny przegląd funkcji gniazd, stanowiący punkt wyjścia do różnicowania oprogramowania klienta i serwera oraz gniazd UDP od połączonych gniazd TCP.

4.3.1. Funkcja `socket`

Gniazda umożliwiają wykonywanie operacji wejścia/wyjścia w komunikacji sieciowej. Tworzymy je za pomocą funkcji `socket`.

Listing 4.1. Funkcja `socket`

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

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

Przy poprawnym wywołaniu funkcja zwraca niewielką nieujemną liczbę całkowitą — deskryptor gniazda, a -1 w przypadku błędu. Argument `domain` określa rodzinę protokołów, najczęściej będzie to dziedzina internetowa w wersji 4 lub 6 lub też dziedzina lokalna, reprezentowane odpowiednio w stałych `AF_INET`, `AF_INET6` oraz `AF_LOCAL`. Argument `type` umożliwia podanie rodzaju gniazda: strumieniowego dla protokołu TCP, datagramowego dla UDP i surowego dla bezpośredniego dostępu do sieci — `SOCK_STREAM`, `SOCK_DGRAM` i `SOCK_RAW`. Argument `protocol` najczęściej ustawiany jest na wartość 0 i oznacza wtedy użycie domyślnego protokołu dla danego typu gniazda w danej rodzinie protokołów. W przypadku gniazd datagramowych można podać jawnie wartość 17 lub użyć stałej `IPPROTO_UDP`. Ustawienie wartości protokołu jest konieczne w przypadku niejednoznaczności, przykładowo dla gniazd surowych.

Więcej zdefiniowanych i obecnie zrozumiałych dla systemu stałych Czytelnik znajdzie na stronie podręcznika systemowego `socket(2)` oraz w [56], rozdziały 4, 14, 25 i 26. Nazwy i numery protokołów dostępne są w pliku `/etc/protocols` i przy użyciu funkcji `getprotoent(3)`. Historycznie do użycia w funkcji `socket` przeznaczone były stałe z przedrostkiem `PF_` (ang. *protocol family*), niemniej ze względu na równoważność ze stałymi z przedrostkiem `AF_` (ang. *address family*), używanymi w gniazdowych strukturach adresowych, proponuje się używać tych drugich.

4.3.2. Funkcja `bind`

Funkcja `bind` dowiązuje do gniazda lokalny adres protokołowy. Dla protokołów internetowych składa się on z adresu IP i numeru portu. Nadanie

adresu umożliwia uzyskanie dostępu do gniazda w danej dziedzinie (lokalnej lub internetowej) i jest wykonywane najczęściej dla gniazd serwerowych[1].

Listing 4.2. Funkcja `bind`

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

3 int bind (int sockfd,
          struct sockaddr *my_addr,
5         socklen_t addrlen);
```

Deskryptor `sockfd` wskazuje na gniazdo, któremu po udanym wywołaniu funkcji `bind` zostaje przypisany adres protokołowy `my_addr`, przekazywany poprzez wskaźnik w drugim argumencie funkcji. Trzeci argument, `addrlen`, określa rozmiar gniazdowej struktury adresowej. W przypadku rodziny gniazd internetowych adres protokołowy umieszczany jest w praktyce w strukturze `sockaddr_in` i rzutowany w wywołaniu funkcji do struktury `sockaddr`. Obydwie te struktury omówione zostały w rozdziale 3.1.3 na str. 43.

Funkcja w przypadku poprawnego wywołania zwraca wartość zero. Wartość -1 zwracana jest najczęściej przy próbie dowiązania już zajętego adresu[2].

Aby dane gniazdo internetowe dostępne było dla dowolnego z interfejsów sieciowych stacji, możemy posłużyć się tzw. *adresem uogólnionym*. W protokole IPv4 dostępny jest on jako stała `INADDR_ANY` (o wartości 0), w IPv6 — `in6addr_any`. Obydwie zdefiniowane są w bibliotece `<netinet/in.h>`.

4.3.3. Funkcje `recvfrom` i `sendto`

Omawiane tutaj funkcje służą do odbierania i wysyłania danych przy użyciu gniazda UDP. Można je traktować jako odpowiedniki standardowych funkcji `read` i `write`.

Listing 4.3. Funkcje `recvfrom` i `sendto`

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

3 ssize_t recvfrom(int sockfd,
          void *buf,
5         size_t len,
          int flags,
7         struct sockaddr *from,
```

[1] Omówienie wywołania funkcji `bind` dla programu klienta można znaleźć np. w [28].
[2] Więcej informacji na ten temat Czytelnik znajdzie w podrozdziale 7.2.1 przy omawianiu opcji `SO_REUSEADDR`.

```
            socklen_t *fromlen);
 9
    ssize_t sendto(int sockfd,
11          const void *buf,
            size_t len,
13          int flags,
            const struct sockaddr *to,
15          socklen_t tolen);
```

Pierwsze trzy argumenty: `sockfd`, `buf` oraz `len` określają kolejno deskryptor gniazda, wskaźnik do bufora zawierającego dane oraz liczbę pobranych i odesłanych bajtów, odpowiednio.

Argument `flags` umożliwia przekazanie do jądra systemu tak zwanych sygnalizatorów, które mogą zmodyfikować działanie funkcji dla pojedynczej operacji wejścia/wyjścia. W typowej wymianie komunikatów między programami klienta i serwera parametr ten ma wartość zero. Więcej informacji na temat możliwych wartości argumentu `flags` i ich zastosowań znajduje się w podrozdziale 7.2.3.

Ostatnie dwa argumenty służą do przekazywania adresu, z którego bądź do którego przesyłane są dane. Ponieważ funkcja `recvfrom` umieszcza w strukturze adresowej `from` adres protokołowy nadawcy datagramu, a w parametrze `fromlen` jego rozmiar, to argumenty te traktowane są jako wyniki funkcji i muszą być przekazywane przez wskaźnik.

Obydwie funkcje jako wartość zwracają liczbę bajtów pobranych lub wysłanych danych. Zauważmy, że 0 jest poprawną wartością w obu przypadkach i nie oznacza to błędu, a jedynie transmisję datagramu zawierającego wyłącznie nagłówki.

Zauważmy również, że w wywołaniu funkcji `recvfrom` można pominąć pobieranie adresu, z którego pochodzi datagram, poprzez ustawienie wartości ostatnich dwóch argumentów na `NULL`. Oznacza to, ze nie interesuje nas adres protokołowy nadawcy.

4.3.4. Uwaga o funkcji `connect`

Mimo braku połączenia między klientem a serwerem pracującymi na gniazdach datagramowych[3], można ustalić w wywołaniu funkcji `connect` z jakim serwerem chcemy się komunikować. Adres podany w argumencie funkcji będzie domyślnym adresem dla wysyłanych datagramów i jedynym, z którego datagramy będą odbierane. Klient UDP może wywoływać funkcję `connect` wielokrotnie w trakcie działania programu.

[3] Funkcja `connect`, (ang. *połącz*), jest funkcją stosowaną typowo dla gniazd strumieniowych.

Użycie funkcji `connect` w przypadku gniazd UDP pozwala także wykryć błędy zgłaszane odpowiednim komunikatem ICMP (patrz podrozdział 2.2.2).

Szczegółowe omówienie parametrów wywołania funkcji Czytelnik znajdzie w podrozdziale 5.3.1.

4.4. Przykład — usługa echo

Standardowa usługa echo dostępna jest na porcie nr 7 zarówno dla gniazd strumieniowych, jak i datagramowych. Polega ona na odsyłaniu przez serwer pojedynczych wierszy otrzymanych od klienta w niezmienionej postaci. Komunikację kończy znak końca pliku. Na przykładzie tej usługi omówimy najistotniejsze aspekty programowania sieciowego dla gniazd UDP.

4.4.1. Program serwera

Listing 4.4. Program echodgs.c

```
 1 #include <sys/types.h>
   #include <sys/socket.h>
 3 #include <stdio.h>
   #include <stdlib.h>
 5 #include <netinet/in.h>
   #include <string.h>
 7
   #define BUFSIZE 1024
 9
   int main(int argc, char **argv)
11 {

13   int sockfd, n;
     struct sockaddr_in addr, clientaddr;
15   uint16_t port;
     socklen_t addrlen;
17   char buf[BUFSIZE];

19   if (argc != 2) {
       fprintf(stderr, "Użycie:␣%s␣port\n", argv[0]);
21     exit(EXIT_FAILURE);
     }
23
     /* tworzymy gniazdo */
25   sockfd = socket(AF_INET, SOCK_DGRAM, 0);
     if (sockfd < 0) {
27     perror("Nieudane␣wywołanie␣socket");
       exit(EXIT_FAILURE);
```

```
29    }

31    port = atoi(argv[1]);

33    /* dowiązanie nazwy */
      addrlen = sizeof(addr);
35    bzero((char *) &addr, addrlen);
      addr.sin_family = AF_INET;
37    addr.sin_port = htons(port);
      addr.sin_addr.s_addr = htonl(INADDR_ANY);
39    if (bind(sockfd, (struct sockaddr *) &addr,
          addrlen)
41         < 0) {
        perror("Nieudane␣wywołanie␣bind");
43      exit(EXIT_FAILURE);
      }
45
      /* odbieranie i odsyłanie datagramów */
47    addrlen = sizeof(clientaddr);
      while (1) {
49      bzero(buf, BUFSIZE);
        n = recvfrom(sockfd, buf, BUFSIZE, 0,
51                   (struct sockaddr *) &clientaddr,
                     &addrlen);
53      if (n < 0) {
          perror("Nieudane␣wywołanie␣recvfrom");
55        exit(EXIT_FAILURE);
        }
57      n = sendto(sockfd, buf, strlen(buf), 0,
                   (struct sockaddr *) &clientaddr,
59                 addrlen);
        if (n < 0) {
61        perror("Nieudane␣wywołanie␣sendto");
          exit(EXIT_FAILURE);
63      }
      }
65  }
```

Zwróćmy uwagę na kolejne etapy programowania serwera.

W naszym programie zakładamy, że numer portu podawany jest w parametrze wywołania. Jeśli chcemy, żeby usługa, którą implementujemy, dostępna była dla użytkowników dłużej niż np. na krótki czas testów, to możemy rozważyć podstawienie pod numer portu pewnej wartości stałej.

Adres dowiązany do gniazda za pomocą funkcji `bind` jest w naszym przypadku adresem uogólnionym.

```
38      addr.sin_addr.s_addr = htonl(INADDR_ANY);
```

Oznacza to, że serwer będzie dostępny w dowolnym interfejsie sieciowym komputera, w tym również przez adres pętli zwrotnej `localhost` (w IPv4 najczęściej 127.0.0.1).

Odbieranie i odsyłanie datagramów do klienta realizowane jest w pętli nieskończonej[4]. Zauważmy, że może zajść sytuacja, kiedy serwer jednocześnie obsługuje wielu klientów, mimo, że powyższy program jest przykładem serwera iteracyjnego. Jest to możliwe dzięki nieutrzymywaniu połączenia między stronami komunikacji. Pakiety przychodzące traktowane są przez program serwera jednakowo i niezależnie od siebie, a dzięki identyfikacji adresu klienta odpowiedzi trafiają do właściwego nadawcy.

Powyższy serwer można testować za pomocą programu `nc` uruchomionego w trybie UDP. Przykładowy test może wyglądać następująco.

```
matrix:~$ ./echodgs 5567 &
 [1] 5287
matrix:~$ nc -u localhost 5567
ala ma kota
ala ma kota


123
123
^C
matrix:~$
```

Jak widać w powyższym przykładzie, program `nc` nie obsługuje poprawnie znaku końca pliku, stąd mało elegancko zakończyliśmy komunikację z serwerem przez przerwanie. Umieszczony poniżej program klienta nie będzie miał już tej wady.

4.4.2. Program klienta

Listing 4.5. Program echodgc.c

```
1 #include <sys/types.h>
  #include <sys/socket.h>
3 #include <stdio.h>
  #include <stdlib.h>
5 #include <netinet/in.h>
```

[4] Zgodnie z ideą serwera świadczenia usługi klientom.

```
#include <string.h>
#include <netdb.h>

#define BUFSIZE 1024

int main(int argc, char **argv)
{

  int sockfd, n;
  struct sockaddr_in addr;
  int port;
  socklen_t addrlen;
  char buf[BUFSIZE];
  char *hostname;
  struct hostent *server;

  if (argc != 3) {
    fprintf(stderr, "Użycie:␣%s␣host␣port\n",
            argv[0]);
    exit(EXIT_FAILURE);
  }

  /* tworzymy gniazdo */
  sockfd = socket(AF_INET, SOCK_DGRAM, 0);
  if (sockfd < 0) {
    perror("Nieudane␣wywołanie␣socket");
    exit(EXIT_FAILURE);
  }

  hostname = argv[1];
  port = atoi(argv[2]);

  /* ustalenie nazwy DNS serwera */
  server = gethostbyname(hostname);
  if (server == NULL) {
    fprintf(stderr,
            "Nie␣ma␣serwera␣o␣nazwie␣%s\n",
            hostname);
    exit(EXIT_FAILURE);
  }
  addrlen = sizeof(addr);
  bzero((char *) &addr, addrlen);
  addr.sin_family = AF_INET;
  addr.sin_port = htons(port);
  bcopy((char *) server->h_addr,
        (char *) &addr.sin_addr.s_addr,
        server->h_length);
```

```
53
   /* wczytywanie/wypisywanie
55  *  i wysyłanie/odbieranie datagramów
    */
57 while (fgets(buf, BUFSIZE, stdin) != NULL) {
     if (feof(stdin)) {
59     exit(EXIT_SUCCESS);
     }
61   n = sendto(sockfd, buf, strlen(buf), 0,
                (struct sockaddr *) &addr, addrlen);
63   if (n < 0) {
       perror("Nieudane␣wywołanie␣sendto");
65     exit(EXIT_FAILURE);
     }
67   n = recvfrom(sockfd, buf, BUFSIZE, 0,
                  (struct sockaddr *) &addr,
69                &addrlen);
     if (n < 0) {
71     perror("Nieudane␣wywołanie␣recvfrom");
       exit(EXIT_FAILURE);
73   }
     fputs(buf, stdout);
75 }
   return EXIT_SUCCESS;
77 }
```

W powyższym programie sprawdzamy, czy podana w argumentach wiersza poleceń nazwa serwera jest poprawną nazwą hosta. Działa to także, gdy poda się zamiast nazwy adres IP (porównaj str. 48).

Oprócz pobierania wierszy z klawiatury i wypisywania odpowiedzi serwera na ekran, jedyną różnicą w przebiegu komunikacji między omawianymi programami klienta i serwera jest kolejność wywołania funkcji `recvfrom` i `sendto`. Niestety, mimo, że protokół komunikacyjny w przypadku usługi echo jest jednoznaczny, może dojść do zablokowania się klienta na wywołaniu funkcji `recvfrom` przykładowo, gdy serwer nie działał w momencie uruchomienia klienta lub odpowiedź serwera zaginęła. Obsługa tego problemu polega na ustawieniu czasu oczekiwania dla funkcji `recvfrom` i może zostać zrealizowana w programie klienta przez obsługę sygnału `SIGALRM`, wywołanie funkcji `select`[5] lub ustawienie dla gniazda opcji `SO_RCVTIMEO`, omówionej w podrozdziale 7.2.2.

Istotną kwestią, z punktu widzenia programowania sieciowego, jest to, że w programie klienta nie ustalamy nigdzie numeru portu związanego z gniaz-

[5] Więcej informacji o użyciu funkcji `select` w programowaniu sieciowym można znaleźć w rozdziale 8.

dem. Istotnie, to jądro systemu dobiera port efemeryczny dla gniazda datagramowego przy pierwszym wywołaniu funkcji `sendto`. Jak już wspominaliśmy, wywołanie funkcji `bind` w programie klienta zdarza się niezwykle rzadko. Jeśli numer portu gniazda klienta byłby ogólnie znany, należałoby w programie dodatkowo sprawdzać, czy przychodząca odpowiedź faktycznie pochodzi od adresata naszego datagramu, zwracając przy tym szczególną uwagę na serwery posiadające więcej niż jeden adres IP.

4.5. Właściwości protokołu — kiedy i jak używać gniazd UDP

Niewątpliwą zaletą protokołu UDP jest prostota i zwięzłość implementacji. Programy pracujące na gniazdach datagramowych są czytelne, a ewentualna obsługa zawodności protokołu ujęta jest jawnie w programie. Przypomnijmy, że do wad protokołu należy możliwość utraty bądź zamiany kolejności pakietów, także ich zdublowanie. Brak wbudowanych mechanizmów kontroli może powodować znaczne przeciążenie sieci, przykładowo przy datagramowej transmisji mediów strumieniowych.

Protokół UDP udostępnia rozgłaszanie i rozsyłanie grupowe. Dzięki swojej specyfice najlepiej nadaje się do pracy w trybie żądanie-odpowiedź i realizuje komunikację bez dodatkowych nakładów, przy najmniejszej możliwej liczbie przesyłanych pakietów. Szybka transmisja danych umożliwia wykorzystanie gniazd datagramowych w aplikacjach pracujących w czasie rzeczywistym. Zaleca się jednakże, aby programy użytkowe korzystające z gniazd UDP zapewniały dodatkowo w programie klienta:

- ustawienie czasu oczekiwania i ponowną transmisję utraconych datagramów,
- numerowanie pakietów i kontrolę kolejności żądań i odpowiedzi.

4.6. Pytania i zadania

1. Napisz serwer UDP usługi daytime (standardowy działa na porcie 13).
2. Napisz programy klienta i serwera, testujące komunikację za pomocą protokołu UDP. Klient powinien wysłać do serwera zadanego w argumencie (nazwa lub numer IP) pewną liczbę (jeżeli nie podano w argumencie to 2000) datagramów zawierających pewną ustaloną liczbę (jeśli nie zadano w argumencie to 1400) bajtów. Jedynym zadaniem serwera będzie odbiór i zliczanie datagramów, a po zakończeniu działania (np. Ctrl-C) wyświetlenie na ekranie ich liczby. Program przetestuj przy połączeniu sieciowym i lokalnym (adres 127.0.0.1).

3. W rozwiązaniu poprzedniego zadania zmodyfikuj program klienta umieszczając w jego kodzie przed wysłaniem datagramu wywołanie funkcji `printf`. Czy zmieni to liczbę odebranych pakietów? Przeanalizuj podobną zmianę wprowadzoną w programie serwera.

Rozdział 5

Gniazda klienckie TCP

5.1. Krótkie wprowadzenie **68**
5.2. Schemat komunikacji procesów klienta i serwera TCP . **68**
5.3. Podstawowe funkcje gniazd klienckich TCP **68**
5.3.1. Funkcja `connect` 70
5.3.2. Funkcje wejścia/wyjścia 70
5.4. Przykład — klient usługi czasu dobowego **71**
5.5. Pytania i zadania **74**

5.1. Krótkie wprowadzenie

Transmission Control Protocol, w skrócie TCP, czyli *Protokół Sterowania Transmisją*, jest najczęściej używanym protokołem sieciowym warstwy transportowej. Szczegółowe omówienie właściwości tego protokołu Czytelnik znajdzie w rozdziale 2.

Przypomnijmy tutaj jedynie, że w protokole TCP tworzone jest połączenie między klientem i serwerem, a komunikacja realizowana jest wraz z mechanizmami zapewniającymi niezawodność przesyłania danych, kontrolę kolejności przesyłanych pakietów oraz sterowanie przepływem uniemożliwiające przepełnienie bufora odbiorczego. Dzięki powyższym właściwościom gniazda utworzone w protokole TCP, nazywane w odróżnieniu do gniazd datagramowych *gniazdami strumieniowymi*, zapewniają niezawodność programowanych usług sieciowych.

5.2. Schemat komunikacji procesów klienta i serwera TCP

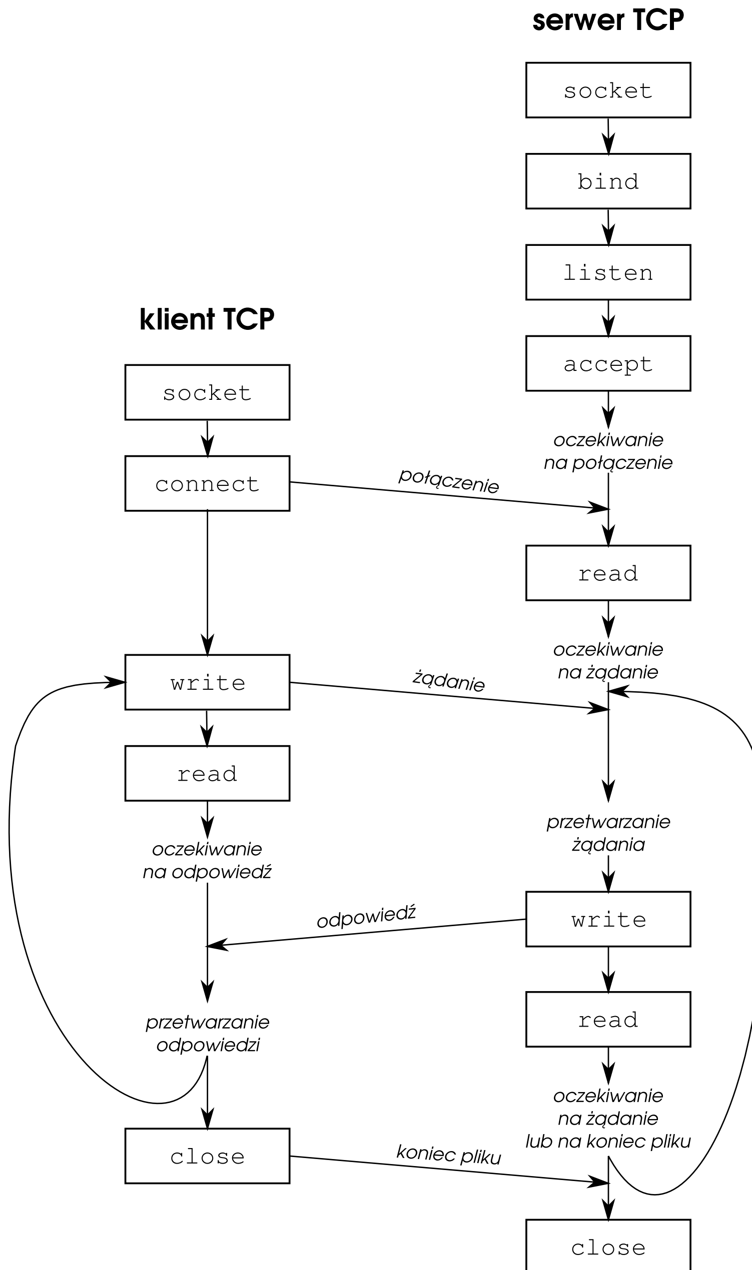
Na rysunku 5.1 przedstawiony został schemat komunikacji między procesami klienta i serwera w protokole TCP. Zakładamy, że serwer został uruchomiony jako pierwszy i oczekuje na zgłoszenie klienta. Gniazdo serwera, utworzone za pomocą funkcji `socket`, służy w tym przypadku jedynie do przyjmowania nadchodzących połączeń. Aby uzyskać taki stan gniazda, konieczne jest wywołanie funkcji `bind`, umożliwiającej nadanie adresu dla gniazda, a następnie funkcji `listen`, która utworzy gniazdo nasłuchujące[1].

Gniazdo klienta łączy się z oczekującym na połączenia serwerem za pomocą funkcji `connect`. Komunikacja między klientem a serwerem realizowana jest najczęściej za pomocą funkcji `recv` i `send`, służących odpowiednio do odbierania i wysyłania danych, ale można równoważnie wykorzystać standardowe funkcje wejścia/wyjścia `read` i `write`. Zauważmy, że bez ustawienia dodatkowych opcji gniazd na obu końcach transmisji może dojść do zablokowania się jednego lub obu programów na którejś z operacji wejścia/wyjścia. W większości przypadków wymiana danych odbywa się według z góry określonego protokołu komunikacyjnego, który określa przykładowo kolejność i dopuszczalną formę żądań klienta oraz możliwych odpowiedzi serwera. Projektowanie protokołów będzie przedmiotem rozdziału 10.

5.3. Podstawowe funkcje gniazd klienckich TCP

Wywołanie funkcji `socket` dla gniazd strumieniowych jest analogiczne jak dla gniazd datagramowych, patrz podrozdział 4.3.1. Typ gniazda

[1] Zobacz str. 79.



Rysunek 5.1. Schemat typowej wymiany komunikatów TCP (porównaj też rysunek 2.6 na stronie 33)

SOCK_STREAM, przekazywany w drugim argumencie, jest poprawny w dziedzinach internetowych AF_INET, AF_INET6 oraz w dziedzinie lokalnej AF_LOCAL. W naszym przykładzie użyliśmy domyślnego protokołu dla gniazd strumieniowych (przez przekazanie wartości 0 w trzecim argumencie wywołania funkcji), można jednakże podać jawnie wartość IPPROTO_TCP.

5.3.1. Funkcja connect

Program klienta, dla wcześniej utworzonego gniazda TCP, wywołuje funkcję connect, aby ustanowić połączenie z serwerem.

Listing 5.1. Funkcja connect

```
1 #include <sys/types.h>
  #include <sys/socket.h>
3
  int connect(int sockfd,
5         const struct sockaddr *serv_addr,
          socklen_t addrlen);
```

W parametrach funkcji przekazywane są kolejno: deskryptor gniazda sockfd, otrzymany w wyniku wywołania funkcji socket, wskaźnik do gniazdowej struktury adresowej serv_addr, zawierającej adres IP i numer portu serwera oraz rozmiar tej struktury addrlen.

Funkcja connect inicjuje uzgadnianie trójfazowe omówione w podrozdziale 2 i zwraca wartość 0 po udanym nawiązaniu połączenia. W przypadku błędu zwracana jest wartość -1. Najczęściej spotykane błędy to: przekroczenie dopuszczalnego czasu oczekiwania na połączenie z serwerem, odmowa połączenia, gdy przykładowo serwer nie działa na zadanym porcie lub chce zakończyć połączenie oraz brak dostępności stacji przy problemach z połączeniem internetowym. Jeśli nie uda się ustanowić połączenia, to gniazdo musi zostać zamknięte przez wywołanie funkcji close[2], gdyż nie możemy dla niego ponownie wywołać funkcji connect.

5.3.2. Funkcje wejścia/wyjścia

Zamiast standardowych funkcji read i write do czytania i pisania do gniazda można wykorzystać funkcje recv i send. Są one odpowiednikami operacji wejścia/wyjścia dla gniazd datagramowych: recvfrom i sendto, omówionych w podrozdziale 4.3.3 na stronie 57. W praktyce funkcje omawiane tutaj różnią się wspomnianych wyżej jedynie brakiem parametrów

[2] Zobacz omówienie funkcji close na str. 80.

służących do przekazywania adresu identyfikującego nadawcę lub odpowiednio odbiorcę datagramu.

Listing 5.2. Funkcje `recv` i `send`

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

ssize_t recv(int sockfd,
        void *buf,
        size_t len,
        int flags);

ssize_t send(int sockfd,
        const void *buf,
        size_t len,
        int flags);
```

Argument `flags`, tak jak dla funkcji `recvfrom` i `sendto` pozwala na modyfikację działania funkcji dla pojedynczej operacji wejścia/wyjścia na gnieździe `sockfd`. W typowej wymianie komunikatów między programami klienta i serwera parametr ten ma wartość zero. Możliwe wartości argumentu `flags` i ich zastosowania Czytelnik znajdzie w podrozdziale 7.2.3.

5.4. Przykład — klient usługi czasu dobowego

Standardowa usługa czasu dobowego (daytime) dostępna jest na ogół na porcie nr 13. Działa ona następująco: po zaakceptowaniu połączenia klienta, serwer wysyła do niego informację o dacie i godzinie w formie takiej, jak wynik polecenia systemowego `date`. Po wysłaniu jednego wiersza danych serwer kończy połączenie. Oznacza to, że w programie klienta, bezpośrednio po udanym powrocie z funkcji `connect`, należy jedynie odebrać dane od serwera. Po wypisaniu otrzymanych danych na standardowym wyjściu klient kończy pracę.

Dla uproszczenia kodu w programie przedstawionym w poniższym listingu skorzystaliśmy z usługi daytime udostępnianej na przykład przez serwer `ntp.task.gda.pl`. Ze względów bezpieczeństwa coraz trudniejsze jest znalezienie działających serwerów czasu. Proponujemy dodatkowo przetestowanie wyników przesyłanych przez serwer `time.ien.it`.

Listing 5.3. Program daytimec.c

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

```
  #include <stdio.h>
4 #include <stdlib.h>
  #include <netinet/in.h>
6 #include <string.h>
  #include <netdb.h>
8
  #define BUFSIZE 1024
10
  int main(int argc, char **argv)
12 {

14   int sockfd, n;
     struct sockaddr_in addr;
16   int port;
     socklen_t addrlen;
18   char buf[BUFSIZE];
     char *hostname;
20   struct hostent *server;

22   /* tworzymy gniazdo */
     sockfd = socket(AF_INET, SOCK_STREAM, 0);
24   if (sockfd < 0) {
       perror("Nieudane␣wywołanie␣socket");
26     exit(EXIT_FAILURE);
     }
28
     hostname = "ntp.task.gda.pl";
30   port = 13;

32   /* ustalenie adresu IP serwera */
     server = gethostbyname(hostname);
34   if (server == NULL) {
       fprintf(stderr, "Nie␣ma␣serwera␣o␣nazwie␣%s\n",
36             hostname);
       exit(EXIT_FAILURE);
38   }
     addrlen = sizeof(addr);
40   bzero((char *) &addr, addrlen);
     addr.sin_family = AF_INET;
42   addr.sin_port = htons(port);
     bcopy((char *) server->h_addr,
44         (char *) &addr.sin_addr.s_addr,
           server->h_length);
46
     if (connect(sockfd, (struct sockaddr *) &addr,
48       addrlen)
         != 0) {
```

```
50      perror("Błąd␣połączenia␣connect");
        exit(EXIT_FAILURE);
52    }

54    /* pobranie daty */
      n = recv(sockfd, buf, BUFSIZE, 0);
56    if (n < 0) {
        perror("Nieudane␣wywołanie␣recv");
58      exit(EXIT_FAILURE);
      }
60    if (n == 0) {
        fprintf(stderr,
62              "Serwer␣zakończył␣połączenie\n");
        exit(EXIT_SUCCESS);
64    }
      buf[n] = '\0';
66    fputs(buf, stdout);

68    return EXIT_SUCCESS;
    }
```

Oczywiście samodzielne przetestowanie powyższego kodu jest również możliwe przy wykorzystaniu programu `nc`. Należy jedynie dostosować parametry serwera w kodzie klienta, przykładowo na:

```
      hostname = "matrix.umcs.lublin.pl";
30    port = 5677;
```

Po tych zmianach można uruchomić serwer testowy poleceniem `nc` z opcją nasłuchu na zadanym porcie i wpisać wynik polecenia `date`.

```
matrix:~$ date
nie, 26 lut 2012, 11:24:22 CET
matrix:~$ nc -l -p 5677
nie, 26 lut 2012, 11:24:22 CET
```

Na drugiej stacji uruchamiamy program klienta czasu dobowego i powinniśmy otrzymać od serwera informację o dacie podaną jak wyżej.

```
localhost:~$ ./daytimec
nie, 26 lut 2012, 11:24:22 CET
localhost:~$
```

Działanie serwera kończymy przez przerwanie Ctrl+C.

5.5. Pytania i zadania

1. Napisz program łączący się z zadanym serwerem (pierwszy argument wiersza poleceń — adres IP lub domenowy) na zadanym porcie (drugi argument wiersza poleceń), który wyświetla na standardowym wyjściu wszystkie dane otrzymane od serwera (sam nic nie wysyła). Program kończy działanie po zakończeniu połączenia przez serwer.
2. Program z poprzedniego zadania zmodyfikuj tak, żeby po odczekaniu pięciu sekund bez otrzymania żadnych danych połączenie zostało zamknięte i program zakończył działanie. Wykorzystaj funkcję `select`.
3. Napisz klienta protokołu `finger`. Program może dostać jeden argument wiersza poleceń w jednej z następujących postaci:

   ```
   user@hostname
   user@
   @hostname
   ```

   Tutaj `user` oznacza identyfikator użytkownika, o którym chcemy uzyskać informację (jeżeli `user` nie występuje w argumencie, to chodzi o informację o wszystkich zalogowanych użytkownikach), a `hostname` nazwę hosta, od którego chcemy uzyskać informację (jeżeli nie ma `hostname` w argumencie, to przyjmujemy localhost). Brak argumentu wiersza poleceń oznacza chęć uzyskania informacji o zalogowanych użytkownikach na hoście localhost.
   W celu uzyskania żądanej informacji program nawiązuje połączenie TCP z zadanym hostem na porcie 79. Następnie wysyła mu jeden wiersz zawierający nazwę użytkownika, o którym chcemy uzyskać informację (jeżeli chodzi o wszystkich zalogowanych użytkowników, to wysyłany jest pusty wiersz). Wszystkie dane otrzymane w wyniku zapytania wysyłane są na standardowe wyjście. Wiersz wysyłany przez program musi kończyć się sekwencją dwóch znaków o kodach 13 i 10 (`'\r'` i `'\n'`).

Rozdział 6

Gniazda serwerowe TCP

6.1. Funkcje gniazda serwera na przykładzie usługi echo . . **76**
6.1.1. Funkcja `bind` 78
6.1.2. Funkcja `listen` 79
6.1.3. Funkcja `accept` 79
6.1.4. Funkcja `close` 80
6.1.5. Funkcja `shutdown` 83
6.1.6. Uzyskiwanie adresów gniazda 84
6.2. Rodzaje serwerów TCP **85**
6.3. Pytania i zadania **85**

6.1. Funkcje gniazda serwera na przykładzie usługi echo

W rozdziale 4.4 przedstawiliśmy przykład oprogramowania sieciowego[1] dla gniazd datagramowych, realizującego usługę echo. W programie klienta wiersze danych, wczytywane ze standardowego wejścia, wysyłane są do serwera, a w następnej kolejności odbierane i drukowane na standardowe wyjście. Serwer odbiera dane i odsyła niezmienione do nadawcy.

Załóżmy teraz, że chcielibyśmy zapewnić niezawodność takiej komunikacji. Najprostsze będzie przepisanie oprogramowania na gniazda strumieniowe, korzystające z wbudowanych mechanizmów kontroli protokołu TCP. Na podstawie kodu serwera usługi echo, przedstawionego w poniższym listingu 6.1, omówimy podstawowe funkcje strumieniowych gniazd serwerowych.

Listing 6.1. Program echos.c

```
1  #include <sys/types.h>
   #include <sys/socket.h>
3  #include <stdio.h>
   #include <stdlib.h>
5  #include <netinet/in.h>
   #include <string.h>
7  #include <unistd.h>

9  #define BUFSIZE 1024
   #define LQUEUE 16
11
   /*
13  * funkcja obsługi klienta
    * realizująca usługę echo
15  */
   int serveclient(int clsockfd)
17 {

19   int n;
     char buf[BUFSIZE];
21
     bzero(buf, BUFSIZE);
23   n = recv(clsockfd, buf, BUFSIZE, 0);
     if (n < 0) {
25     perror("Nieudane wywołanie recv");
       return EXIT_FAILURE;
27   }
     while (n > 0) {
29     n = send(clsockfd, buf, strlen(buf), 0);
```

[1] Zobacz kody źródłowe programów serwera (listing 4.4) i klienta (listing 4.5), znajdujące się na stronach 59 i 61, odpowiednio.

```
        if (n < 0) {
31        perror("Nieudane␣wywołanie␣send");
          return EXIT_FAILURE;
33      }
        bzero(buf, BUFSIZE);
35      n = recv(clsockfd, buf, BUFSIZE, 0);
      }
37    return EXIT_SUCCESS;
    }
39
    int main(int argc, char **argv)
41  {

43    int sockfd, consockfd;
      struct sockaddr_in addr, clientaddr;
45    uint16_t port;
      socklen_t addrlen;
47
      if (argc != 2) {
49      fprintf(stderr, "Użycie:␣%s␣port\n", argv[0]);
        exit(EXIT_FAILURE);
51    }

53    /* tworzymy gniazdo */
      sockfd = socket(AF_INET, SOCK_STREAM, 0);
55    if (sockfd < 0) {
        perror("Nieudane␣wywołanie␣socket");
57      exit(EXIT_FAILURE);
      }
59
      port = atoi(argv[1]);
61
      /* dowiązanie nazwy */
63    addrlen = sizeof(addr);
      bzero((char *) &addr, addrlen);
65    addr.sin_family = AF_INET;
      addr.sin_port = htons(port);
67    addr.sin_addr.s_addr = htonl(INADDR_ANY);
      if (bind(sockfd, (struct sockaddr *) &addr,
69        addrlen)
          < 0) {
71      perror("Nieudane␣wywołanie␣bind");
        exit(EXIT_FAILURE);
73    }

75    /* gniazdo nasłuchujące i dlugość kolejki */
      if (listen(sockfd, LQUEUE) < 0) {
```

```
77      perror("Nieudane␣wywolanie␣listen");
        exit(EXIT_FAILURE);
79    }

81    addrlen = sizeof(clientaddr);
      while (1) {
83      if ((consockfd =
             accept(sockfd, (struct sockaddr *) &addr,
85                  &addrlen)) < 0) {
          perror("Nieudane␣wywolanie␣accept");
87        exit(EXIT_FAILURE);
        }
89
        /* obsługa klienta na gnieździe połączonym */
91      if (serveclient(consockfd) < 0) {
          fprintf(stderr,
93          "Błąd␣komunikacji␣w␣funkcji␣serveclient");
        }
95      close(consockfd);
      }
97  }
```

6.1.1. Funkcja `bind`

Dowiązanie do utworzonego za pomocą funkcji `socket` gniazda `sockfd` adresu protokołowego zostało już omówione w podrozdziale 4.3.2. Przypomnijmy, że funkcja `bind` jest typowa dla gniazd serwerowych, które mają być dostępne na danym komputerze poprzez ogólnie znany port.

W przypadku gniazd TCP nie musimy obowiązkowo wypełniać żadnego z pól adresu protokołowego — zostaną one automatycznie dobrane przez jądro systemu w momencie wywołania funkcji `connect` lub `listen`. Jeśli nie ustalimy adresu IP, to będą przyjmowane połączenia skierowane do dowolnego z interfejsów sieciowych danej stacji[2]. Jeśli podamy zerowy numer portu, to jądro dobierze również port efemeryczny dla danego gniazda. Poza wyjątkiem serwerów RPC nie ma to większego zastosowania, ponieważ serwery są rozpoznawane dzięki ich dobrze znanym numerom portów. Chcąc otrzymać wartość przydzielonego przez jądro systemu efemerycznego numeru portu, musimy wywołać funkcję `getsockname`, przekazującą adres protokołowy, omówioną szerzej na str. 84.

[2] Porównaj pojęcie adresu uogólnionego omówione na str. 57.

6.1.2. Funkcja `listen`

Funkcja `listen` jest funkcją wywoływaną jedynie przez serwer TCP.

Listing 6.2. Funkcja `listen`

```
1 #include <sys/types.h>
  #include <sys/socket.h>
3
  int listen(int sockfd, int backlog);
```

Zadaniem powyższej funkcji jest przekształcenie gniazda `sockfd` do stanu biernego, definiowanego jako gotowość do przyjmowania nowych połączeń. Takie gniazdo będziemy nazywać *gniazdem nasłuchującym*. Zwróćmy uwagę na fakt, że za pomocą takiego gniazda nigdy nie są przekazywane dane.

Drugi argument funkcji `listen` podaje maksymalną ilość połączeń, które system ustawia w kolejce do gniazda. W rzeczywistości dla danego gniazda nasłuchującego tworzone są dwie kolejki. Jedna zawiera połączenia znajdujące się w trakcie nawiązywania, druga — już nawiązane[3]. Argument `backlog` określa maksymalną wartość łącznej liczby elementów obydwu tych kolejek. W wielu przykładach, z powodu historycznych ograniczeń systemowych, wciąż używa się liczby 5 jako wartości tego argumentu, mimo, że w praktyce liczba ta jest zbyt mała wobec dzisiejszego obciążenia serwerów, które obsługują miliony połączeń dziennie. W omawianym przez nas programie serwera definiujemy długość kolejki za pomocą makrodefinicji

```
10 #define LQUEUE 16
```

zakładając, że będzie to wielkość wystarczająca dla usługi echo.

6.1.3. Funkcja `accept`

Mając gniazdo nasłuchujące, za pomocą wywołania funkcji `accept`, serwer TCP może przyjmować przychodzące połączenia.

Listing 6.3. Funkcja `accept`

```
1 #include <sys/types.h>
  #include <sys/socket.h>
3
  int accept(int sockfd,
5           struct sockaddr *addr,
```

[3] Gniazda klientów, dla których zakończono uzgadnianie trójfazowe, znajdują się w stanie `ESTABLISHED`, a podczas uzgadniania, kiedy wysyłany jest segment `SYN`, stan gniazda określany jest jako `SYN_RCVD`, patrz podrozdział 2.3.1.2.

```
            socklen_t *addrlen);
```

W parametrach wynikowych `addr` oraz `addrlen` przekazywany jest adres protokołowy połączonego klienta. Informację tą można wykorzystać do diagnostyki połączeń po stronie serwera lub zignorować poprzez podstawienie wskaźników pustych.

Zaakceptowane połączenie klienta wybierane jest jako pierwsze z kolejki połączeń nawiązanych utworzonej przez system dla gniazda nasłuchującego `sockfd`. Wynikiem zwracanym przez funkcję, o ile nie wystąpi błąd, jest deskryptor *gniazda połączonego*, przydzielonego automatycznie dla tego klienta przez jądro systemu. W omawianym przez nas przykładzie serwera usługi echo deskryptor gniazda połączonego zostanie przekazany przez funkcję `accept` do zmiennej `consockfd`.

```
consockfd =
accept (sockfd,
        (struct sockaddr *) &addr,
        &addrlen)
```

Zauważmy, że przez cały okres działania serwera istnieje na ogół tylko jedno gniazdo nasłuchujące, podczas gdy gniazd połączonych będzie tyle, ile zaakceptowanych połączeń.

Omawiany przykład implementuje serwer iteracyjny, stąd dopóki klient prowadzi z serwerem wymianę komunikatów przy użyciu gniazda `consockfd`, realizowaną w pomocniczej funkcji `serveclient`, dopóty nie jest możliwe przyjęcie nowego połączenia na gnieździe nasłuchującym `sockfd`. W podrozdziale 6.2 omawiamy pozostałe typy serwerów TCP i sposoby ich realizacji.

6.1.4. Funkcja `close`

Standardowej funkcji `close` możemy używać również do zamykania deskryptorów gniazd.

Listing 6.4. Funkcja `close`

```
#include <unistd.h>

int close(int fd);
```

Wywołanie tej funkcji oznacza, że na danym gnieździe nie będą już wykonywane żadne operacje na poziomie aplikacji, niemniej, jeśli w buforze nadawczym pozostały jakieś dane, to warstwa TCP spróbuje je wysłać i dopiero

wtedy spróbuje zakończyć połączenie. Można zmodyfikować to zachowanie poprzez odpowiednie ustawienie opcji SO_LINGER[4].

Bardzo istotne jest kontrolowanie liczby odwołań do gniazda połączonego w przypadku serwerów współbieżnych, gdzie zarówno proces macierzysty jak i potomny mają kopię deskryptora. Wywołanie funkcji close jedynie w procesie potomnym nie spowoduje faktycznego zakończenia połączenia, tylko zmniejszenie liczby odwołań do gniazda. Ponadto proces macierzysty, jeśli nie zamyka gniazd utworzonych w funkcji accept, może szybko doprowadzić do wyczerpania puli dostępnych dla jednego procesu otwartych deskryptorów.

Funkcja close kończy omawiany tutaj przykład serwera TCP usługi echo. Przywołajmy ponownie program nc i użyjmy go do przetestowania kodu zaprezentowanego w listingu 6.1.

```
matrix:~$ ./echos 5667 &
[1] 3316
matrix:~$ nc localhost 5667
a
a
bb
bb
^C
matrix:~$
```

Poniżej dołączamy kod klienta usługi echo pracującego na gniazdach strumieniowych, który pozwoli na zakończenie komunikacji z serwerem przez wpisanie znaku końca pliku Ctrl+D.

Listing 6.5. Program echoc.c

```
1 #include <sys/types.h>
  #include <sys/socket.h>
3 #include <stdio.h>
  #include <stdlib.h>
5 #include <netinet/in.h>
  #include <string.h>
7 #include <netdb.h>

9 #define BUFSIZE 1024

11 int main(int argc, char **argv)
   {
13
     int sockfd, n;
```

[4] Więcej informacji na ten temat można znaleźć w rozdziale 7.5 w [56].

```
15    struct sockaddr_in addr;
      int port;
17    socklen_t addrlen;
      char buf[BUFSIZE];
19    char *hostname;
      struct hostent *server;
21
      if (argc != 3) {
23      fprintf(stderr, "Użycie:␣%s␣host␣port\n",
                argv[0]);
25      exit(EXIT_FAILURE);
      }
27
      /* tworzymy gniazdo */
29    sockfd = socket(AF_INET, SOCK_STREAM, 0);
      if (sockfd < 0) {
31      perror("Nieudane␣wywołanie␣socket");
        exit(EXIT_FAILURE);
33    }

35    hostname = argv[1];
      port = atoi(argv[2]);
37
      /* ustalenie adresu IP serwera */
39    server = gethostbyname(hostname);
      if (server == NULL) {
41      fprintf(stderr, "Nie␣ma␣serwera␣o␣nazwie␣%s\n",
                hostname);
43      exit(EXIT_FAILURE);
      }
45    addrlen = sizeof(addr);
      bzero((char *) &addr, addrlen);
47    addr.sin_family = AF_INET;
      addr.sin_port = htons(port);
49    bcopy((char *) server->h_addr,
            (char *) &addr.sin_addr.s_addr,
51          server->h_length);

53    if (connect(sockfd, (struct sockaddr *) &addr,
                  addrlen)
55        != 0) {
        perror("Błąd␣połączenia␣connect");
57      exit(EXIT_FAILURE);
      }
59
      /* wczytywanie/wypisywanie
61     *  i wysyłanie/odbieranie wierszy tekstu
```

```
     */
63   while (fgets(buf, BUFSIZE, stdin) != NULL) {
       if (feof(stdin)) {
65       exit(EXIT_SUCCESS);
       }
67     n = send(sockfd, buf, strlen(buf), 0);
       if (n < 0) {
69       perror("Nieudane␣wywołanie␣send");
         exit(EXIT_FAILURE);
71     }
       n = recv(sockfd, buf, BUFSIZE, 0);
73     if (n < 0) {
         perror("Nieudane␣wywołanie␣recv");
75       exit(EXIT_FAILURE);
       }
77     if (n == 0) {
         fprintf(stderr,
79               "Serwer␣zakończył␣połączenie\n");
         exit(EXIT_SUCCESS);
81     }
       fputs(buf, stdout);
83   }
     return EXIT_SUCCESS;
85 }
```

6.1.5. Funkcja `shutdown`

Funkcję `shutdown` wywołujemy w momencie, kiedy chcemy, aby dany proces natychmiast zainicjował zakończenie połączenia.

Listing 6.6. Funkcja `shutdown`

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

3 int shutdown(int sockfd, int how);
```

W odróżnieniu od funkcji `close`, próbującej zamknąć połączenie w obu kierunkach, funkcji `shutdown` można użyć do zakończenia jednego typu operacji: czytania lub pisania do gniazda `sockfd`. Może to znacznie przyspieszyć całą operację zamykania połączenia dzięki uniknięciu niepotrzebnego oczekiwania na przesłanie ewentualnych danych, które i tak zostałyby odrzucone. Funkcja ta, co jest bardzo istotne, spowoduje zakończenie połączenia nawet w przypadku, gdy licznik odwołań do deskryptora jest większy od zera.

Parametr `how` może przyjąć jedną z trzech wartości: `SHUT_RD` (0), `SHUT_WR` (1) oraz `SHUT_RDWR` (2) oznaczające kolejno: zamknięcie części czytającej połączenia, zamknięcie części piszącej (nazywane *półzamknięciem*) i w końcu zamknięcie obydwu części. W przypadku półzamknięcia dane znajdujące się w buforze wysyłkowym gniazda są wysyłane przed zainicjowaniem sekwencji zamykającej połączenie TCP. Zamknięcie części czytającej gniazda powoduje odrzucenie wszystkich odebranych, ale nieprzeczytanych danych.

6.1.6. Uzyskiwanie adresów gniazda

Funkcje `getsockname` i `getpeername` umożliwiają uzyskanie adresu protokołowego związanego z gniazdem, za pomocą którego komunikujemy się z innym procesem, oraz adresu gniazda znajdującego się na drugim końcu tej komunikacji, odpowiednio.

Listing 6.7. Funkcje `getsockname` i `getpeername`

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

3 int getsockname(int sockfd,
          struct sockaddr *name,
5         socklen_t *namelen)

7 int getpeername(int sockfd,
          struct sockaddr *name,
9         socklen_t *namelen);
```

Obie funkcje umieszczają wynikowy adres w gniazdowej strukturze adresowej wskazywanej parametrem `name` i zwracają w przypadku poprawnej operacji wartość zero, a −1 w przeciwnym razie.

Funkcji `getsockname` możemy chcieć użyć przykładowo w programie klienta dla lokalnego gniazda połączonego zwróconego przez funkcję `connect`. W programie serwera, jeśli funkcja `bind` nie określała jawnie składowych dowiązywanego adresu protokołowego[5], można również wywołać funkcję `getsockname`, aby poznać dokładne wartości przypisane przez jądro systemu.

W szczególnych sytuacjach programowania serwerów współbieżnych, korzystających z systemowej funkcji `exec`, może się zdarzyć, że jedynym sposobem serwera na uzyskanie adresu klienta jest wywołanie funkcji `getpeername` dla gniazda połączonego.

[5] Sytuacja taka ma miejsce dla adresu uogólnionego i/lub zerowego numeru portu, zobacz uwagi na stronach 57 i 78.

6.2. Rodzaje serwerów TCP

Omawiany w tym rozdziale przykład dotyczył prostego serwera iteracyjnego, w którym obsługa klienta połączonego blokowała przyjmowanie kolejnych połączeń. W skrajnej sytuacji, gdy klient przez dłuższy czas nie podejmuje próby zakończenia komunikacji, dochodzi wręcz do zablokowania działania usługi echo.

Pewien rodzaj współbieżności serwera można wprowadzić za pomocą funkcji `select`, dzięki której serwer może obsługiwać jednocześnie pewną liczbę klientów. Mimo prostoty idei, nie jest to jednak rozwiązanie doskonałe, ze względu na ograniczoną wydajność i kłopotliwość w programowaniu — w pewnych przypadkach.

Standardowe podejście do współbieżności polega na tworzeniu nowego procesu obsługującego klienta za pomocą funkcji `fork`, podczas gdy proces macierzysty niezależnie kontynuuje oczekiwanie i akceptuje kolejne połączenia na gnieździe nasłuchującym (w językach wyższego poziomu podejście to może być realizowane za pomocą wątków). Więcej informacji na ten temat Czytelnik znajdzie w rozdziale 8.

6.3. Pytania i zadania

1. Program z serwera z listingu 6.1 zmodyfikuj tak, żeby wyświetlał na standardowym wyjściu adres IP klienta, jego nazwę domenową oraz informację, który raz połączono się z tego adresu.
2. Napisz program umożliwiający rozmowę przez sieć pomiędzy dwoma użytkownikami. Program może być uruchomiony z jednym lub dwoma argumentami wiersza poleceń. Jeżeli uruchomiony jest z jednym parametrem (serwer), to oznacza on port, na którym program nasłuchuje na połączenie; jeżeli z dwoma (klient), to oznaczają one adres komputera (IP lub domenowy) i numer portu, z którym program ma się połączyć. Po nawiązaniu (lub zaakceptowaniu) połączenia programy na zmianę wysyłają wiadomości wczytane ze standardowego wejścia (po jednym wierszu) i wyświetlają na standardowym wyjściu odpowiedź. Pierwsza wiadomość wysyłana jest przez klienta. Program kończy działanie, gdy ze standardowego wejścia wczytany zostanie koniec pliku lub rozmówca zamknie połączenie.

Rozdział 7

Opcje gniazd

7.1. Sterowanie opcjami gniazd **88**
  7.1.1. Funkcje `getsockopt` i `setsockopt` 88
  7.1.2. Funkcje `fcntl` i `ioctl` 90
7.2. Przegląd najważniejszych opcji **91**
  7.2.1. Ogólne opcje gniazd 91
    `SO_ERROR` 91
    `SO_KEEPALIVE` 92
    `SO_RCVBUF`, `SO_SNDBUF` 92
    `SO_REUSEADDR` 92
  7.2.2. Wejście/wyjście nieblokujące 93
    `SO_RCVTIMEO`, `SO_SNDTIMEO` 93
  7.2.3. Sygnalizatory metod `send`, `sendto`, `recv` i `recvfrom` 93
7.3. Pytania i zadania **94**

7.1. Sterowanie opcjami gniazd

Opcje umożliwiają modyfikację zachowania gniazd w pewnych określonych sytuacjach. Możemy skorzystać z funkcji pobierających i ustawiających opcje typowe dla deskryptorów gniazd, czyli z `getsockopt` i `setsockopt`, lub z ogólniejszych funkcji systemowych, `fcntl` i `ioctl`, działających na dowolnych plikach.

Odpowiednie ustawienie opcji gniazda może nie tylko udostępnić pewne dodatkowe funkcjonalności, ale także przyspieszyć działanie oprogramowania i poprawić jego skuteczność.

7.1.1. Funkcje `getsockopt` i `setsockopt`

Funkcje `getsockopt` i `setsockopt` mają zastosowanie wyłącznie do gniazd.

Listing 7.1. Funkcje `getsockopt` i `setsockopt`

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

3 int getsockopt(int sockfd,
           int level,
5          int optname,
           void *optval,
7          socklen_t *optlen);

9 int setsockopt(int sockfd,
           int level,
11         int optname,
           const void *optval,
13         socklen_t optlen);
```

Argument `sockfd` określa deskryptor gniazda otwartego, a `level` — warstwę sieciową, na poziomie której dana opcja ma zastosowanie. Najczęściej stosowane wartości `level` to `SOL_SOCKET`, odpowiadająca ogólnemu oprogramowaniu gniazd, oraz `IPPROTO_IP` i `IPPROTO_TCP` dla oprogramowania właściwego danemu protokołowi. Argument `optname` określa nazwę opcji. Wskaźnik `optval` umożliwia przekazanie zmiennej, w której ma zostać zapisana, lub z której ma być pobrana wartość opcji. Rozmiar tej zmiennej określa parametr `optlen`, który w przypadku funkcji `getsockopt` jest argumentem-wynikiem. W przypadku, gdy dana opcja jest sygnalizatorem danej właściwości, argument `optval` zawsze powinien wskazywać wartość 0 przy jej wyłączaniu, a wartość różną od zera przy włączaniu.

Obydwie funkcje zwracają 0 w przypadku poprawnego wywołania, a −1, jeśli wystąpił błąd.

Przykładowo, wykorzystanie omawianych funkcji może polegać na podwojeniu rozmiaru bufora odbiorczego dla gniazda datagramowego.

```
1  int sockfd, n, size;
   socklen_t len;
3
   sockfd = socket(AF_INET, SOCK_DGRAM, 0);
5  if (sockfd < 0) {
     perror("Nieudane␣wywołanie␣socket");
7    exit(EXIT_FAILURE);
   }
9
   len = sizeof(size);
11 n = getsockopt(sockfd, SOL_SOCKET, SO_SNDBUF,
                  (void *)&size, &len);
13 size = size * 2;
   n = setsockopt(sockfd, SOL_SOCKET, SO_SNDBUF,
15                (void *)&size, sizeof(size));
```

Ustawienie własności gniazda pokażemy na przykładzie opcji umożliwiającej ponowne wykorzystanie lokalnego adresu protokołowego przez funkcję `bind`.

```
1  int sockfd, n, on;
   struct sockaddr_in addr;
3
   sockfd = socket(AF_INET, SOCK_STREAM, 0);
5  if (sockfd < 0) {
     perror("Nieudane␣wywołanie␣socket");
7    exit(EXIT_FAILURE);
   }
9
   on = 1;
11 n = setsockopt(sockfd, SOL_SOCKET, SO_REUSEADDR,
                  (void *)&on, sizeof(on));
13
   addrlen = sizeof(addr);
15 bzero((char *) &addr, addrlen);
   addr.sin_family = AF_INET;
17 addr.sin_port = htons(PORT);
   addr.sin_addr.s_addr = htonl(INADDR_ANY);
19
   if (bind(sockfd, (struct sockaddr *) &addr, addrlen)
21     < 0) {
```

```
    perror("Nieudane wywołanie bind");
23  exit(EXIT_FAILURE);
  }
```

7.1.2. Funkcje `fcntl` i `ioctl`

Funkcje `fcntl` i `ioctl` umożliwiają wykonywanie różnych operacji na deskryptorze pliku, które dla gniazd dotyczą m.in. ustanawiania i odczytywania właściciela oraz ustawiania dla gniazda trybu nieblokującego wejścia/wyjścia lub wejścia/wyjścia sterowanego sygnałami.

Funkcja `ioctl` udostępnia też inne zaawansowane operacje dla programów administracyjnych takich jak `ifconfig` i `route`. Zainteresowanych odsyłamy do rozdziałów 16 i 17 w [56].

Funkcja `fcntl` jest preferowana dla wykonywania wspomnianych na początku podstawowych operacji na gniazdach.

Listing 7.2. Funkcja `fcntl`

```
  #include <unistd.h>
2 #include <fcntl.h>

4 int fcntl(int sockfd, int cmd);
  int fcntl(int sockfd, int cmd,
6           long arg);
  int fcntl(int fd, int cmd, struct flock *lock);
```

Podając w argumencie `cmd` wartości `F_GETOWN` i `F_SETOWN` możemy odpowiednio odczytać i ustawić właściciela gniazda `sockfd` na identyfikator procesu lub grupy procesów przekazany w argumencie `arg`. Jest to przydatne w przypadku gniazda, którego wejścia/wyjścia sterowane jest sygnałami oraz gdy gniazdo odbiera dane pozapasmowe.

Za pomocą polecenia `F_SETFL` przekazanego w argumencie `cmd` możemy ustawić gniazdo nieblokujące, przekazując w argumencie `arg` sygnalizator `O_NONBLOCK`. Podobnie, ustawiony dla gniazda sygnalizator `O_ASYNC` spowoduje generowanie sygnału `SIGIO` w chwili, gdy zmienia się stan gniazda.

Ważne jest, aby przy ustawianiu i wyłączaniu powyższych sygnalizatorów nie zapomnieć, że gniazdo, tak jak każdy inny plik w systemie, może mieć ustawiony szereg innych znaczników. Poniżej przedstawiamy przykład, jak może wyglądać wywołanie `fcntl` ustawiające gniazdo w tryb nieblokujący, nie zmieniające pozostałych sygnalizatorów.

```
1 int sockfd, flags;
```

```
3 /* ustanowienie gniazda nieblokującego */
  if ((flags = fcntl(sockfd, F_GETFL, 0)) < 0) {
5   perror("Błąd␣F_GETFL");
    exit(EXIT_FAILURE};
7 }

9 flags |= O_NONBLOCK;

11 if (fcntl(sockfd, F_SETFL, flags) < 0)
    perror("Błąd␣F_SETFL");
13  exit(EXIT_FAILURE};
  }
```

Wyłączenie tego samego znacznika można zrealizować zamieniając wiersz nr 9 na:

```
flags &= ~O_NONBLOCK;
```

7.2. Przegląd najważniejszych opcji

Listę wszystkich opcji gniazd wraz z programem testującym ich dostępność w danym systemie operacyjnym można znaleźć w rozdziale 7 w [56]. Tutaj omówimy kilka spośród najczęściej stosowanych w typowych programach użytkowych.

Zauważmy, że pewne opcje należy ustawiać zanim na gnieździe będą wykonywane odpowiednie operacje, przykładowo wejścia/wyjścia. Inną istotną własnością jest dziedziczenie przez gniazdo połączone TCP opcji gniazda nasłuchującego. Dlatego, jeśli chcemy dla gniazda połączonego ustawić którąkolwiek z opcji: `SO_DEBUG`, `SO_DONTROUTE`, `SO_KEEPALIVE`, `SO_LINGER`, `SO_OOBINLINE`, `SO_RCVBUF` oraz `SO_SNDBUF`, musimy ustanowić ją w rzeczywistości dla gniazda nasłuchującego.

7.2.1. Ogólne opcje gniazd

`SO_ERROR`

Błędy gniazda nieodczytane (ang. *pending errors*) jeszcze ze zmiennej `so_error` mogą w szczególnych sytuacjach spowodować, że wynik kolejnej operacji na gnieździe zostanie niewłaściwie przekazany do systemowej zmiennej `errno`. Odczytanie opcji `SO_ERROR` pozwala uzyskać informację o ostatnim błędzie gniazda. Może to być przydatne przy nieblokującym wywołaniu

funkcji `connect`, w celu sprawdzenia, czy faktycznie nawiązano połączenie oraz przy rozwiązywaniu problemów z odczytem z gniazda.

Opcji tej nie można ustawiać za pomocą funkcji `setsockopt`. Przeczytanie jej zawartości w wywołaniu funkcji `getsockopt` spowoduje odczytanie błędu z `so_error` i ustawienie wartości tej zmiennej na 0.

SO_KEEPALIVE

Włączenie opcji `SO_KEEPALIVE` dla gniazda TCP powoduje, że w przypadku dłuższego braku przesyłania danych między połączonymi gniazdami do partnera wysyłany jest pusty segment. Jeśli nie nadejdzie potwierdzenie (segment ACK) dla tego segmentu lub odebrany zostanie komunikat o błędzie, to gniazdo zostanie zamknięte. W przeciwnym wypadku działanie procesu nie jest przerywane.

Opcja `SO_KEEPALIVE` najczęściej używana jest w oprogramowaniu serwera w celu wykrycia półotwartego połączenia z klientem, którego stacja załamała się i od którego dane już nigdy nie nadejdą. Na poziomie warstwy TCP można wykorzystać opcję `TCP_KEEPALIVE` do zmiany domyślnego maksymalnego czasu oczekiwania na aktywność połączenia.

SO_RCVBUF, SO_SNDBUF

Powyższe opcje umożliwiają zmianę domyślnego rozmiaru buforów odbiorczego i wysyłkowego, odpowiednio, dla danego gniazda.

Bufor odbiorczy przechowuje pobrane dane, zanim zostaną one odczytane przez program użytkowy. Protokół TCP steruje przepływem danych, co oznacza, że nadawca nie może wysłać więcej danych niż zaoferowany przez odbiorcę rozmiar bufora. Konsekwencją tej właściwości protokołu jest, że ewentualna zmiana domyślnych rozmiarów obydwu buforów musi zostać wykonana przed wywołaniami funkcji `connect` u klienta i `listen` w programie serwera.

W przypadku protokołu UDP przychodzący datagram, który nie mieści się w buforze odbiorczym, jest odrzucany, także łatwo może dojść do utraty datagramów u wolniejszego odbiorcy, jeśli nadawca wysyła swoje stosunkowo szybko. Z tego powodu rozmiar bufora wysyłkowego dla gniazd datagramowych powinien być sporo mniejszy od rozmiaru bufora odbiorczego.

SO_REUSEADDR

Opcja `SO_REUSEADDR` jest niezwykle ważna dla pracy serwerów, a ustawienie jej umożliwia ponowne wykorzystanie lokalnego adresu protokołowego przez funkcję `bind`.

Przykładowo można to wykorzystać do wznowienia pracy serwera na-

słuchującego[1], w którym obsługa klienta odbywa się w procesie potomnym przez istniejące połączenie. Innym zastosowaniem jest umożliwienie pracy kilku egzemplarzy tego samego serwera na tym samym porcie, o ile każdy z nich dowiązuje inny adres lokalny IP[2]. Ponadto, ustawienie opcji `SO_REUSEADDR` może być przydatne w początkowym testowaniu działania serwera, gdyż umożliwia przydzielenie tego samego numeru portu nawet, gdy jądro systemu nie dołączyło go ponownie do puli dostępnych portów tuż po zakończeniu działania poprzedniego procesu.

Bez włączenia tej opcji w powyższych sytuacjach funkcja `bind` zwraca błąd *„Address already in use”*.

7.2.2. Wejście/wyjście nieblokujące

Przypomnijmy, że w podrozdziale 7.1.2, dotyczącym zastosowania funkcji `fcntl` dla gniazd, podaliśmy przykład ustawiania gniazda w tryb nieblokujący za pomocą sterowania ogólnymi znacznikami pliku. Inną możliwością jest ustawienie odpowiednich opcji gniazda.

`SO_RCVTIMEO`, `SO_SNDTIMEO`

Powyższe opcje stosujemy do jednorazowego ustawienia czasu oczekiwania dla gniazda, obowiązującego we wszystkich operacjach wysyłania i pobierania danych. Domyślnie czas oczekiwania jest wyłączony.

Dzięki zastosowaniu `SO_RCVTIMEO` możliwe jest zapobieganie zablokowaniu się gniazda na wywołaniu funkcji `recv` czy `recvfrom`. Wartość maksymalnego dopuszczalnego czasu oczekiwania umieszczamy w strukturze `timeval`, podobnie jak w funkcji `select`, a funkcja odpowiadająca danej operacji czytania w przypadku jego przekroczenia zwraca błąd `EWOULDBLOCK`. Podobnie `SO_SNDTIMEO` odpowiada za ustawienie czasu oczekiwania, aby zapobiec ewentualnemu zablokowaniu gniazda wykonującego operację pisania.

Zwróćmy uwagę, że żadna z opcji nie umożliwia ustawienia czasu oczekiwania dla funkcji `connect`. Problem ten można rozwiązać za pomocą funkcji `select` lub wprowadzając czas oczekiwania za pomocą obsługi sygnału `SIGALRM`.

7.2.3. Sygnalizatory metod `send`, `sendto`, `recv` i `recvfrom`

Dołączamy w tym miejscu informację o sygnalizatorach modyfikujących zachowanie pojedynczych operacji wejścia/wyjścia dla gniazd. Nie są to w istocie opcje, ponieważ mają zastosowanie w pojedynczym wywołaniu funk-

[1] Wznowienie odbywa się przez ponowne wywołanie kolejno funkcji `socket`, `bind` i `listen`.

[2] Jest to zachowanie typowe dla komputerów posiadających wiele aliasów adresu IP.

cji, niemniej wpływają na pracę gniazda w podobnym (choć węższym) zakresie.

W wywołaniu funkcji `send`, `sendto`, `recv` i `recvfrom`, można ustawić czwarty w kolejności argument na jedną z wartości zebranych w tabeli 7.1 lub alternatywę bitową ich większej ilości.

Tabela 7.1. Zestawienie sygnalizatorów przekazywanych w argumencie `flags`

Sygnalizator	Znaczenie	Funkcje `send`, `sendto`	Funkcje `recv`, `recvfrom`
`MSG_DONTROUTE`	stacja docelowa znajduje się w sieci lokalnej, niepotrzebne wyszukiwanie w tabeli tras	•	
`MSG_DONTWAIT`	pojedyncza operacja nieblokująca	•	•
`MSG_PEEK`	podgląd nadchodzącego komunikatu, bez faktycznego odebrania danych		•
`MSG_WAITALL`	czytanie określonej ilości danych, funkcja czeka na otrzymanie wszystkich		•
`MSG_EOR`	koniec rekordu logicznego (nie dotyczy TCP)	•	
`MSG_OOB`	wysłanie/odebranie priorytetowych danych pozapasmowych	•	•

Zwróćmy uwagę, że sygnalizator `MSG_DONTROUTE` może zastąpić w wywołaniu funkcji `send` ustawienie ogólnej opcji gniazda `SO_DONTROUTE`. W przypadku włączenia opcji `SO_OOBINLINE`, która powoduje, że dane pozapasmowe umieszczane są bezpośrednio w kolejce danych wejściowych, nie można używać sygnalizatora `MSG_OOB`.

7.3. Pytania i zadania

1. Napisz program, który dla gniazda TCP odczyta i wyświetli na standardowym wyjściu wartości najważniejszych opcji gniazd.
2. W rozwiązaniu zadania 2 z rozdziału 4 (str. 64) zmniejsz dwukrotnie rozmiar bufora odbiorczego serwera. Co zaobserwujesz?
3. Zmodyfikuj rozwiązanie zadania 2 z rozdziału 6 (str. 85) tak, aby maksymalny czas oczekiwania serwera na dane klienta wynosił 5 minut.

Rozdział 8

Demony i usługi

8.1. Procesy w Uniksie **96**
8.2. Sygnały **99**
8.3. Multipleksacja wejścia/wyjścia **100**
8.4. Serwery współbieżne **103**
8.5. Demony **105**
8.6. Pytania i zadania **108**

8.1. Procesy w Uniksie

Żeby w Uniksie efektywnie tworzyć serwery usług musimy zapoznać się jeszcze z kilkoma pojęciami, których zastosowanie to umożliwia. Pierwszym z nich jest *proces* — czyli wczytany do pamięci operacyjnej i uruchomiony przez system operacyjny program.

Do zarządzania procesami służą przede wszystkim funkcje następujące:

```
#include <unistd.h>
    pid_t fork(void);
    void _exit(int status);
#include <stdlib.h>
    void exit(int status);
#include <sys/types.h>
#include <unistd.h>
    pid_t getpid(void);
    pid_t getppid(void);
#include <sys/types.h>
#include <sys/wait.h>
    pid_t wait(int *status);
    pid_t waitpid(pid_t pid, int *status,
                  int options);
#include <sys/types.h>
#include <sys/time.h>
#include <sys/resource.h>
#include <sys/wait.h>
    pid_t wait3(int *status, int options,
                struct rusage *rusage);
    pid_t wait4(pid_t pid, int *status, int options,
                struct rusage *rusage);
#include <unistd.h>
    int execve(const char *filename,
               char *const argv[],
               char *const envp[]);
    int execl(const char *path,
              const char *arg, ...);
    int execlp(const char *file,
               const char *arg, ...);
    int execle(const char *path, const char *arg,
               ..., char * const envp[]);
    int execv(const char *path, char *const argv[]);
    int execvp(const char *file,
               char *const argv[]);
```

Podstawową funkcją jest tu `fork`, która tworzy nowy proces. Proces ten jest klonem procesu wywołującego i nawet swoje działanie rozpoczyna od

miejsca wywołania tej funkcji. Dziedziczy po procesie-rodzicu prawie wszystkie atrybuty (takie, jak cały kod, otwarte pliki[1], dane, stan zmiennych, uprawnienia itp.) — dostaje jednak, co oczywiste, własny unikalny identyfikator (można sprawdzić go funkcją `getpid`[2]) i tożsamość. Mamy więc do momentu wywołania funkcji `fork` jeden proces, a bezpośrednio po powrocie z tej funkcji już dwa. Program może rozpoznać, w którym z procesów jest przez wartość zwróconą z tej funkcji: w procesie-dziecku zwrócone zostaje 0, a w procesie-rodzicu — identyfikator utworzonego procesu-dziecka.

Sama funkcja `fork` może się nie powieść z różnych powodów — zwraca wtedy wartość -1 i nie tworzy nowego procesu.

Proces może zakończyć sam siebie funkcją biblioteczną `exit` albo funkcją systemową `_exit`. Pierwsza z tych funkcji (powszechnie znana, patrz też str. 5) powinna zapewnić należyte zamknięcie wszystkich otwartych plików/strumieni oraz opróżnienie ich buforów i wykonanie innych czynności kończących. W normalnych warunkach funkcja ta daje więc pewność, że nie utracimy danych. Inaczej jest z funkcją `_exit`, która kończy po prostu proces i już.

Proces-rodzic jest w pewnym sensie odpowiedzialny za powołane do życia procesy-dzieci, więc powinien poczekać na ich zakończenie i odebrać od nich status zakończenia programu. Służy do tego cała rodzina funkcji `wait`. Wszystkie z tych czterech funkcji zwracają identyfikator procesu-dziecka[3], który się zakończył do momentu zakończenia wykonywania danej funkcji — o ile jeszcze nie był taki proces-dziecko obsłużony przez funkcję z rodziny `wait`. Jeśli takich procesów jest kilka, to obsługiwany w jednym wywołaniu jest jeden z nich, na pozostałe trzeba 'poczekać' osobno.

Funkcja `waitpid` może zostać wywołana z różnymi wartościami parametru `pid`, z których najbardziej przydatne są:

— `== -1` oznaczające oczekiwanie na dowolne dziecko;
— `> 0` oznaczające oczekiwanie na dziecko o podanym identyfikatorze `pid`.

Jeśli `status == NULL`, to jest ignorowany, a w przeciwnym razie musi być adresem, pod który zostanie wstawiony status zakończenia procesu, wraz z innymi informacjami. Można je odzyskać przy pomocy poniższych (między innymi) makr:

— `WIFEXITED(status)` zwraca odpowiedź na pytanie, czy proces zakończył się normalnie (przez zwykłe zakończenie jego funkcji `main` lub przez wywołanie funkcji `exit` lub `_exit`; tylko jeśli jest to prawdą, można użyć

[1] To szczególnie przyda się w podrozdziale 8.4.
[2] A identyfikator rodzica — funkcją `getppid`.
[3] Albo, oczywiście, -1 na oznaczenie błędu.

makra `WEXITSTATUS(status)` do odzyskania wartości przekazanej przez funkcję/instrukcję kończącą proces;

— `WIFSIGNALED(status)` zwraca odpowiedź na pytanie, czy proces został zakończony sygnałem (patrz podrozdział 8.2); tylko jeśli tak się stało, można użyć makr `WTERMSIG(status)` (zwraca numer owego sygnału) oraz `WCOREDUMP(status)` (odpowiada na pytanie, czy zakończenie procesu spowodowało jego zrzut rdzenia pamięci — *core dumped*)[4].

W końcu ostatni parametr, `options` może przyjmować wartości oznaczające różne opcje, z których (poza wartością `0`, oznaczającą brak specjalnych opcji) najważniejszą jest `WNOHANG`, która powoduje, że funkcja nie czeka na zakończenie procesu, tylko od razu wraca, biorąc pod uwagę tylko ewentualne procesy zakończone (a nieobsłużone) przed jej wywołaniem. Jeśli takowych brak, zwracane jest `0`.

Wywołanie `wait(buf_stat)` jest równoważne wywołaniu `waitpid(-1, buf_stat, 0)`, natomiast funkcje `wait3` oraz `wait4` oferują po prostu dodatkowe informacje o użytych przez proces-dziecko zasobach systemu.

Nie byłoby wiele pożytku z nowych procesów, gdyby nie można było wykonywać w nich nowego kodu. Rodzina funkcji `exec` służy właśnie do załadowania nowego kodu. Kod załadowany przez którąś z tych funkcji **zastępuje** całkowicie kod i dane procesu wywołującego, nie zmienia jednak jego tożsamości (w tym jego identyfikatora czy też uprawnień).

Pierwszym parametrem w tych funkcjach jest zawsze nazwa programu do uruchomienia, przy czym w tych funkcjach, które mają w nazwie literę `p`, nazwa (`file`) będzie poszukiwana w ścieżkach zawartych w zmiennej środowiskowej `PATH` (tak, jak to się dzieje w powłoce). W pozostałych funkcjach musimy podać pełną (aczkolwiek może być względna) nazwę ścieżkową (`filename`, `path`).

Litera `e` na końcu nazwy funkcji oznacza z kolei, że przekazujemy jawnie środowisko w parametrze `envp`, w formacie takim, jak zmienna `environ` (str. 3). W końcu litera `l` lub `v` oznacza sposób przekazywania argumentów wiersza poleceń nowemu programowi — odpowiednio w postaci listy kolejnych argumentów funkcji (`arg` i następne) lub tablicy napisów (`argv` — całkiem analogicznie jak parametr `argv` funkcji `main`). Uwaga: w obu przypadkach ostatnim argumentem musi być pusty wskaźnik `NULL`, a pierwszym — nazwa ścieżkowa programu!

[4] Makro `WCOREDUMP` ma jeszcze jedno ograniczenie: trzeba się upewnić, że jest zdefiniowane, bo nie należy do standardu POSIX. Jest jednak implementowane w wielu systemach, w tym także w Linuksie.

8.2. Sygnały

Sygnał jest w systemie operacyjnym odpowiednikiem przerwania i umożliwia bardzo prymitywną komunikację między procesami. Pośrednikiem dostarczającym sygnały (a często także ich nadawcą) jest jądro systemu operacyjnego. Z sygnałami związane są między innymi poniższe funkcje.

```
#include <sys/types.h>
#include <signal.h>
    int kill(pid_t pid, int sig);
#include <signal.h>
    int sigprocmask(int how,
                    const sigset_t *set,
                    sigset_t *oldset);
    sighandler_t signal(int signum,
                        sighandler_t handler);
```

Funkcja `kill` służy do przesłania procesowi o podanym identyfikatorze `pid`[5] sygnału o numerze `sig`. W drugim parametrze najlepiej posługiwać się stałymi zdefiniowanymi w pliku nagłówkowym `signal.h`, których używać będziemy i tutaj. Najczęściej spotykane sygnały to:

— `SIGTERM` wysyłany jest w celu zakończenia procesu; może jednak być (jak większość sygnałów) zablokowany, przechwycony, zignorowany (patrz niżej);
— `SIGKILL` wysyłany jest także w celu zakończenia procesu, jednakże odbiorca nie może go przechwycić ani zignorować i zostaje automatycznie i natychmiast przez jądro zakończony;
— `SIGINT` całkiem podobny do `SIGTERM`, jednakże nie jest zwykle wysyłany funkcją `kill`, ale przez wciśnięcie odpowiednich klawiszy (zwykle `Ctrl-C`) na terminalu, na którym proces-odbiorca działa;
— `SIGHUP` jest sygnałem normalnie wysyłanym do procesu, gdy jego terminal sterujący (patrz podrozdział 8.5) zawiesi się (na przykład przez zamknięcie okna terminala); sygnał ten jest jednak wykorzystywany (po przechwyceniu) przez demony (które nie mają terminala sterującego) do ich restartu;
— `SIGQUIT` jak `SIGINT` (też z klawiatury, zwykle `Ctrl-\`), ale z dodatkowym zrzutem pamięci (*core dumped*);
— `SIGSTOP` wstrzymanie procesu;
— `SIGTSTP` wstrzymanie procesu z terminala;
— `SIGCONT` kontynuacja procesu po wstrzymaniu.

[5] O ile ten parametr jest dodatni — w przeciwnym razie znaczenie tego parametru jest nieco inne.

Jak widać z powyższej listy, domyślna reakcja procesu na sygnał może być różna. W ogólności może być to:

— zakończenie procesu[6];
— zakończenie procesu ze zrzutem rdzenia pamięci;
— zignorowanie sygnału;
— wstrzymanie procesu;
— kontynuacja wstrzymanego procesu (tylko jeden sygnał: `SIGCONT`).

Sygnały można blokować i odblokowywać („maskować" — służy do tego funkcja `sigprocmask`), a także zmieniać ich domyślną obsługę, do czego służy funkcja `signal`[7]. Wyjątkiem są dwa sygnały (`SIGKILL` oraz `SIGSTOP`), których nie można zablokować, a ich obsługi nie można zmienić.

Funkcja `signal` w pierwszym parametrze `signum` dostaje numer sygnału, którego obsługa ma być zmieniona, natomiast drugi parametr `handler` to adres funkcji, która ma stanowić nową obsługę danego sygnału. Jako wartość parametru `handler` można także użyć stałych `SIG_IGN` (która powoduje, że sygnał będzie ignorowany) oraz `SIG_DFL` (która przywraca domyślną obsługę sygnału).

Wartością zwracaną przez `signal` jest adres funkcji stanowiącej poprzednią obsługę sygnału (lub też `SIG_IGN` albo `SIG_DFL`), a w przypadku błędu: `SIG_ERR`.

Sam typ parametru `handler` (a tym samym wyniku funkcji `signal`) jest zdefiniowany jako:

```
typedef void (*sighandler_t)(int);
```

Tak więc funkcja obsługująca sygnał niczego ma nie zwracać, a dostaje jeden parametr — którym jest numer sygnału.

8.3. Multipleksacja wejścia/wyjścia

Multipleksacja (zwielokrotnianie) wejścia/wyjścia to oczekiwanie na gotowość do odczytu lub zapisu wielu deskryptorów jednocześnie. Oczekiwanie na gotowość jednego deskryptora jest realizowana po prostu funkcjami `write`

[6] Najczęstszy efekt — i stąd zapewne nazwa funkcji systemowej…

[7] Opisujemy tutaj tę funkcję, bowiem jest ona prosta w użyciu. Niestety, w różnych systemach jej dokładne działanie różni się nieco, stąd jej przenośność jest wątpliwa. Można zamiast niej użyć funkcji `bsd_signal` (o tym samym interfejsie, co `signal`), która zachowuje się tak, jak w systemach z rodziny BSD (i tak, jak domyślnie `signal` pod Linuksem), ale nie wszędzie jest zdefiniowana. Natomiast zalecane jest użycie funkcji `sigaction`, która jest przenośna, ale nieco bardziej skomplikowana.

oraz `read`, jeśli plik jest otwarty w trybie blokującym (co jest trybem domyślnym, patrz str. 1.3.3). Jednakże, jeśli chcemy testować gotowość kilku deskryptorów, moglibyśmy otworzyć je w trybie nieblokującym i testować na zmianę w pętli `while`, aż któryś będzie gotowy. Niestety, rozwiązanie to jest fatalne ze względu na efektywność — podczas takiej pętli procesor musi aktywnie ją wykonywać. . . !

Na szczęście istnieje rozwiązanie, które usypia proces w oczekiwaniu na gotowość któregoś z podanych deskryptorów (a więc procesor nie musi go obsługiwać, lecz proces jest budzony, gdy zajdzie odpowiednie zdarzenie związane z podanymi deskryptorami), a realizowane jest to przy pomocy funkcji:

```
#include <sys/select.h>
    int select(int nfds, fd_set *readfds,
                         fd_set *writefds,
                         fd_set *exceptfds,
               struct timeval *timeout);
```

Parametrami funkcji `select` są wskaźniki `readfds`, `writefds`, `exceptfds` do trzech zestawów (zbiorów) deskryptorów, które mają być „śledzone”. Pierwszy zestaw (`*readfds`) zawiera deskryptory plików, na których oczekujemy gotowości do odczytu. Drugi zestaw (`*writefds`) to deskryptory plików, na których spodziewamy się gotowości do zapisu. W końcu trzeci zestaw (`*exceptfds`) to deskryptory plików, na których spodziewamy się sytuacji wyjątkowych. Każdy z owych trzech adresów może być równy `NULL`, co oznacza, że nie oczekujemy na dany rodzaj zdarzeń w ogóle. Związany z owymi trzema zestawami jest parametr pierwszy `nfds`, który musi być o `1` większy od najwyższego z deskryptorów występujących w podanych zestawach.

Ostatni z parametrów omawianej funkcji, `timeout` jest wskaźnikiem do struktury zdefiniowanej następująco:

```
struct timeval {
    long tv_sec;
    long tv_usec;
};
```

Pola tej struktury określają limit czas czekania w sekundach (`tv_sec`) oraz w mikrosekundach (`tv_usec`). Tu także może być `timeout == NULL`, co oznacza oczekiwanie bez limitu (może być nieskończone).

Działanie funkcji `select` sprowadza się do uśpienia procesu w oczekiwaniu na zajście jednego (lub kilku) z poniższych zdarzeń:

— gotowość do odczytu na jednym z deskryptorów z zestawu `*readfds`;
— gotowość do zapisu na jednym z deskryptorów z zestawu `*writefds`;
— sytuacja wyjątkowa na jednym z deskryptorów z zestawu `*exceptfds`;
— upłynięcie zadanego czasu.

W takim wypadku kończy się wywołanie funkcji `select`[8] i obiekty wskazywane przez cztery parametry wskaźnikowe zawierają wyniki — oczywiście tylko te, które nie były równe `NULL`. Po zakończeniu `select` zestaw `*readfds` zawiera te z podanych przed wywołaniem deskryptorów, które są gotowe do odczytu; zestaw `*writefds` — te które są gotowe do zapisu; zestaw `*exceptfds` — te które są w stanie wyjątkowym; Ponadto wynikiem zwracanym przez samą funkcję `select` jest sumaryczna liczba deskryptorów ustawiona w tych (wynikowych) zbiorach[9]. Natomiast `timeout` w niektórych[10] implementacjach (np. w Linuksie) zawiera czas pozostały do wykorzystania całego limitu czasu.

Pozostało jeszcze wyjaśnienie, w jaki sposób posługujemy się zbiorami deskryptorów — czyli danymi typu `fd_set`. Interfejs programisty dla tego typu stanowią funkcje[11]:

```
#include <sys/select.h>
    void FD_CLR(int fd, fd_set *set);
    int  FD_ISSET(int fd, fd_set *set);
    void FD_SET(int fd, fd_set *set);
    void FD_ZERO(fd_set *set);
```

Trzech pierwszych funkcji używamy do ustawienia zawartości zbiorów przed (każdorazowym!) wywołaniem funkcji `select`: funkcja `FD_ZERO` czyści podany zbiór `*set`; funkcje `FD_SET` oraz `FD_CLR` odpowiednio dodają i usuwają podany deskryptor `fd` do/ze zbioru `*set`. Natomiast po zakończeniu działania funkcji `select` należy użyć funkcji `FD_ISSET`, która zwraca prawdę (wartość niezerową), wtedy i tylko wtedy, gdy dany deskryptor `fd` należy do podanego zbioru `*set`.

[8] Wywołanie funkcji `select` skończyć się może także błędem — wtedy zwracana jest wartość -1, a wskazywane obiekty mają wartości nieokreślone.

[9] Może to być 0, jeśli limit czasu minął, a nie doszło do żadnego z oczekiwanych wydarzeń na deskryptorach.

[10] Ale nie we wszystkich — nie należy raczej więc z tego korzystać, jeśli chcemy przenośności nasz aplikacji.

[11] Być może zdefiniowane jako makra preprocesora — stąd duże litery.

8.4. Serwery współbieżne

W kontekście niniejszego skryptu, całe dotychczasowe rozważania tego rozdziału zmierzają do tego, byśmy potrafili skonstruować oprogramowanie pozwalające obsługiwać wiele połączeń współbieżnie. Jest to standardowy sposób działania serwerów — chodzi bowiem o to, by mogły one obsługiwać jednocześnie (współbieżnie) wiele połączeń — kolejne bez zamykania poprzednich.

Pierwszy z rodzajów serwerów współbieżnych opiera się na uruchamianiu w miarę potrzeby kolejnych procesów (za pomocą funkcji `fork`), z których każdy odpowiada za obsługę jednego połączenia. Schemat takiego serwera (wraz z komentarzami) przedstawiony jest na listingu 8.1. Odmianą tego rodzaju jest tak zwany serwer *pre-fork*, który uruchamia kilka procesów potomnych po rozpoczęciu nasłuchiwania, ale przed akceptacją połączenia — wtedy każdy z potomków oczekuje niezależnie od pozostałych na połączenie, obsługuje je, a potem może wrócić do nasłuchiwania.

Listing 8.1. Współbieżny serwer wieloprocesowy

```
...

gn0 = socket(...);
bind(gn0, ...);
listen(gn0, ...);
/* powyżej, w każdym wywołaniu, należałoby jeszcze
 * sprawdzić, czy funkcje nie zwróciły wartości
 * oznaczającej błąd; to samo dotyczy poniższtch
 * wywołań accept oraz fork */

while (1) {
    pol = accept(gn0, ...);
    if (fork() != 0) {
        close(pol);
        /* jesteśmy w procesie głównym, więc zamykamy
         * deskryptor związany z połączeniem */
    } else {
        close(gn0)
        /* jesteśmy w procesie potomnym, więc zamykamy
         * deskryptor związany z nasłuchiwaniem */
        ...
        /* a tutaj zajmujemy się obsługą połączenia
         * pol aż do jego zakończenia */
    }
}

...
```

Innym sposobem osiągnięcia współbieżności obsługi połączeń jest wykorzystanie multipleksacji — nie tworzymy nowych procesów dla kolejnych połączeń, ale za pomocą funkcji `select` oczekujemy na dane z któregoś z otwartych połączeń (oraz na nowe połączenia) i w zależności od tego, które gniazdo się akurat odezwie — to jest odpowiednio obsługiwane. Schemat pokazany jest na listingu 8.2.

Listing 8.2. Współbieżny serwer jednoprocesowy

```
...

fd_set gniazda, gn_pom;

/* wszędzie poniżej opuszczamy sprawdzanie błędów
 * wykonania funkcji systemowych
 * -- czego normalnie nie należy oczywiście
 * robić! */

gn0 = socket(...);
bind(gn0, ...);
listen(gn0, ...);

FD_ZERO(&gniazda);
FD_SET(gn0, &gniazda);
max_gn = gn0+1;

while (1) {
  FD_ZERO(&gn_pom);
  for (gn = 0; gn < max_gn; gn++)
    if (FD_ISSET(g, &gniazda))
      FD_SET(g, &gn_pom);
  select(max_gn, &gn_pom, NULL, NULL, NULL);
  for (gn = 0; gn < max_gn; gn++)
    if (FD_ISSET(gn, &gn_pom))
      if (gn == gn0) {
        pol = accept(gn0, ...);
        FD_SET(pol, &gniazda);
        if (pol >= max_gn)
          max_gn = pol+1;
      } else {
        ...
        /* obsługa pojedynczego czytania z gn
         * wraz z odpowiedzią;
         * także ewentualne jego zamknięcie
         * (wtedy należy usunąć gn ze zbioru
         * gniazda za pomocą FD_CLR oraz być może
         * odpowiednio zmniejszyć max_gn */
```

```
39          }
    }
41
    ...
```

Serwery oparte na funkcji `select` mają zastosowanie w sytuacjach, gdy spodziewamy się wielu połączeń, ale ruch na nich jest dość mały. Jeśli natomiast ruch jest duży, a tym bardziej, jeśli mamy do dyspozycji maszynę wieloprocesorową, zalecane są serwery oparte na funkcji `fork`.

8.5. Demony

Na zakończenie niniejszego rozdziału pozostało nam opisanie typowego sposobu uruchamiania usług w systemach Uniksowych, czyli uruchamiania ich jako *demonów*. Demon w Uniksie to proces, który ma pewną specjalną własność, mianowicie: nie posiada *terminala sterującego*.

Każdy proces uruchomiony normalnie z konsoli ma tę konsolę jako terminal sterujący. Oznacza to, że wydarzanie związane z terminalem (takie, jak jego zamknięcie; czy też pewne kombinacje klawiszy wciśnięte przez użytkownika) generują pewne zdarzenia (zwykle sygnały), które wpływają na zachowanie wszystkich procesów, dla których ów terminal jest sterujący.

Z pojęciem terminala sterującego związane są także pojęcia grup procesów i sesji procesów: sesja ma przypisany jeden terminal sterujący, może natomiast zawierać wiele grup, z których każda może z kolei składać się z wielu procesów. Każda grupa i każda sesja może mieć swojego lidera (zwykle jest to tworzący ją proces).

Kiedy nowy proces jest tworzony (`fork`), jego grupa i sesja procesów (oraz terminal sterujący) są takie same, jak procesu-rodzica. Grupę procesów można stworzyć lub zmienić (tylko w obrębie bieżącej sesji!) za pomocą funkcji `setpgid`. Jednakże — ponieważ odbywa się to w ramach sesji, nie powoduje zerwania związku z terminalem sterującym.

Jedynym sposobem zmiany sesji (i tym samym oderwania się od terminala sterującego) jest stworzenie nowej sesji bezparametrową funkcją `setsid`, która działa tylko, jeśli nie wywołuje jej lider grupy procesów. Proces wywołujący tę funkcję tworzy nową sesję pozbawioną terminala sterującego, w której istnieje dokładnie jedna nowa grupa, i jedynym członkiem (i jednocześnie liderem) nowej grupy i nowej sesji jest ów tworzący ją proces.

Oprócz tej, najważniejszej właściwie, czynności proces stający się demonem powinien wykonać zwykle kilka innych, organizacyjnych. Pełny algorytm przekształcania procesu w demona (*demonizacji procesu*) przedstawiony jest na listingu 8.3.

Listing 8.3. Schemat demonizacji procesu

```
...

void fork_exit(void) {
  if ((pid = fork()) < 0) {
    perror("fork");
    _exit(EXIT_FAILURE);
  } else
    if (pid != 0) {
      _exit(EXIT_SUCCESS);
    }
}

int main(int argc char *argv[]) {
  ...
  fork_exit();

  setsid();

  fork_exit();

  chdir("/");
  umask(0);

  close(0);
  close(1);
  close(2);

  open("/dev/null", O_RDWR);
  dup2(0, 1);
  dup2(0, 2);

/* tu już właściwy program... */
```

Listing 8.3 wymaga kilka wyjaśnień, bo każda czynność w jego ramach wykonywana jest celowa, aczkolwiek nie każda jest obowiązkowa.

1. Wiersze 3–11 definiują pomocniczą funkcję, której zadaniem jest uruchomienie nowego procesu potomnego, kontrola błędu i zakończenia procesu rodzica[12]. Czynność ta wykonywana będzie dwukrotnie, stąd definiujemy ją jako osobny podprogram. W tej funkcji mamy także sprawdzanie błędów wykonania funkcji systemowej i odpowiednią reakcję na nie —

[12] Funkcja `_exit` jest tu użyta, żeby uniknąć niepożądanych dodatkowych czynności wykonywanych przez `exit`.

w dalszej części listingu 8.3 opuszczamy to, ale normalnie nie należy tego oczywiście robić.
2. Wiersz 15 tworzy proces-dziecko, po czym kończy pracę procesu-rodzica. Dzięki temu po uruchomieniu naszego programu znowu powłoka się zgłosi, a nowy proces (który pracę kontynuuje) nie będzie liderem grupy (bo należy do tej samej grupy, do której należał wcześniej proces-rodzic i która już lidera miała). Ponadto, dzięki zakończeniu procesu-rodzica, osierocony proces potomny będzie zaadoptowany przez proces `init`, który zajmował się będzie odbieraniem statusu jego zakończenia (i nie będzie procesów-zombie).
3. Wiersz 17 tworzy nową sesję (a także grupę) procesów pozbawioną terminala sterującego. Jest to możliwe, bo po wykonaniu linii 15 mamy pewność, że proces nie jest liderem grupy. Może więc własną sesję utworzyć.
4. Wiersz 19 powtarza stworzenie potomka — a to po to, by nasz demon nie był liderem sesji. W niektórych systemach bowiem, lider sesji po otwarciu urządzenia terminalowego automatycznie przyjmuje je za terminal sterujący (jeśli takiego nie ma). Chcemy naszego demona przed tym zabezpieczyć.
   Kolejne wiersz są właściwie opcjonalne, ale tradycyjnie wykonuje je się, żeby uzyskać standardowe zachowanie programu w często spotykanych sytuacjach i uniknąć różnych kłopotów/niespodzianek.
5. Wiersz 21 zmienia katalog bieżący programu na taki, który na pewno zawsze istnieje i nie zostanie odmontowany — takim katalogiem jest oczywiście katalog główny. Zapobiega to blokowaniu przez demona prób odmontowania innych katalogów.
6. Wiersz 22 likwiduje maskę (`umask`) nowo tworzonych plików — tak, by demon miał pełną kontrolę nad uprawnieniami każdego nowo tworzonego pliku.
7. Wiersze 24–26 zamykają standardowe deskryptory, związane zwykle z terminalem — nie chcemy, żeby nasz demon interferował jakkolwiek z innym oprogramowaniem korzystającym ewentualnie z tego terminala.
8. Natomiast wiersze 28–30 wiążą te standardowe deskryptory z wirtualnym urządzeniem pustym. Alternatywą dla tych wierszy (szczególnie, jeśli chodzi o deskryptor 2) jest otwarcie jakiegoś pliku, gdzie wysyłane będą logi lub inne informacje diagnostyczne — tak, by wszystkie informacje trafiające na standardowe wyjście błędów (na przykład przez użycie funkcji `perror`) nie ginęły, ale były później do analizy.

Dalszą część stanowi już zwykłe działanie serwera. Warto jednak jeszcze być może pamiętać o zmianie działania sygnałów — przynajmniej `SIGHUP` (który dla demonów ma tradycyjnie znaczenie specjalne, patrz str. 99) oraz

SIGTERM (żeby ten sygnał kończył działanie programu, ale przedtem robił ewentualnie porządek (zapisywał informacje, zamykał pliki, opróżniał bufory itp.).

8.6. Pytania i zadania

1. Napisz program, który uruchamia program podany w pierwszym argumencie linii poleceń (korzystając do jego znalezienia ze zmiennej środowiskowej PATH) i przekazuje mu kolejne argumenty. Przed uruchomieniem tego programu wyświetla (na standardowe wyjście diagnostyczne) komunikat o próbie jego uruchomienia (z nazwą programu i jego argumentami), potem czeka na jego zakończenie, a wtedy (znowu na standardowe wyjście diagnostyczne) wyświetla dostępne informacje o jego zakończeniu (status; sygnał, którym został zakończony; informacja, czy zrzucony został rdzeń).
2. Napisz program podobny do tego z poprzedniego zadania, uruchamiający jednak podany program w trybie demona. Możesz przewidzieć dodatkowe argumenty linii poleceń, które będą plikami, do których przekierowane będą standardowe strumienie wejścia/wyjścia (deskryptory 0, 1, 2).
3. Napisz współbieżny serwer usługi echo w dwóch wersjach (wielo- i jednoprocesowej), ale z modyfikacją taką, że połączenie nie jest zamykane. Napisz do celów testowych klienta, który łączy się z takim serwerem raz i z użyciem tego jednego połączenia wysyła co sekundę jakiś tekst, odbiera odpowiedź i ją wyświetla. Przetestuj swoje programy uruchamiając jeden serwer i wiele klientów jednocześnie z różnych sesji lub (lepiej) różnych komputerów.

Rozdział 9

Protokoły warstwy aplikacji

9.1. Poczta elektroniczna **110**
9.1.1. Adres e-mail 111
9.1.2. Format wiadomości pocztowej 112
9.1.2.1. Nagłówek wiadomości 112
9.1.2.2. Ciało wiadomości i MIME 113
9.1.3. Protokoły 115
9.1.3.1. SMTP 116
9.1.3.2. Protokoły odbioru wiadomości: POP3 i IMAP 118
9.1.4. Webmail 122
9.1.5. Bezpieczeństwo poczty 123
9.1.5.1. Szyfrowanie 123
9.1.5.2. Fałszowanie poczty (Fake-mail) . . . 123
9.2. WWW **124**
9.2.1. URL 124
9.2.2. Protokół HTTP 125
9.2.2.1. Pole `Host` i serwery wirtualne 127
9.2.2.2. Połączenia w HTTP 127
9.2.2.3. Zawartość części zasadniczej 128
9.2.2.4. Przekierowania 129
9.2.2.5. Pamięć podręczna (ang. *cache*) . . . 129
9.2.2.6. Sesje w HTTP i ciasteczka 129
9.2.2.7. Uwierzytelnianie 130
9.2.2.8. Kody odpowiedzi 130
9.2.2.9. Metoda POST 131
9.2.2.10. Metoda HEAD 132
9.2.2.11. Różnice między HTTP/1.1, a HTTP/1.0 132
9.3. Pytania i zadania **133**

Protokoły warstwy zastosowań w Internecie korzystają najczęściej z jednego z dwóch protokołów transportowych opisanych przez nas w podrozdziale 2.3 — UDP lub TCP. Różnice między usługami świadczonymi przez nie warstwie aplikacji mają duży wpływ na różnice między protokołami opartymi o UDP, a tymi opartymi na TCP. Protokoły korzystające z UDP (np. protokół DNS) zazwyczaj wymieniają krótkie wiadomości w postaci binarnej — ważny jest rozmiar datagramu. Protokoły oparte o TCP (np. WWW, protokoły pocztowe) zazwyczaj są protokołami, w których dane sterujące są zwykłym tekstem ASCII podzielonym na wiersze oddzielone od siebie sekwencją znaków CRLF — znakiem o kodzie ASCII 13 i znakiem o kodzie ASCII 10[1]. W dalszej części rozdziału słowo wiersz lub linia będzie zawsze oznaczało wiersz zakończony taką właśnie sekwencją. Dane tekstowe ułatwiają testowanie i diagnozowanie problemów związanych z działaniem tych protokołów. Zresztą większość standardowych protokołów tekstowych w przesyłanych przez siebie komunikatach umieszcza krótkie informacje tekstowe, które z założenia są informacjami diagnostycznymi dla czytającego je człowieka. Działanie programu w ogóle od nich nie zależy. Program czasami je po prostu wyświetla, ale na pewno nie powinien na ich podstawie podejmować decyzji co do wykonania jakichś akcji. Do testowania takich protokołów można użyć np. narzędzi takich jak `telnet`, czy `netcat` opisanych w dodatku C.

W dalszej części tego rozdziału omówimy dwie spośród popularnych usług internetowych — *pocztę elektroniczną* (*e-mail*) i sieć *WWW* (ang. World Wide Web).

Skupimy się głównie na omówieniu protokołów warstwy aplikacji realizujących te usługi. W przypadku poczty elektronicznej tymi protokołami są *SMTP* (ang. *Simple Mail Transfer Protocol*), *POP3* (ang. *ang. Post Office Protocol version 3*) i *IMAP* (ang. *ang. Internet Message Access Protocol*). Protokołem, na którym bazuje WWW jest *HTTP* (protokół przesyłania dokumentów hipertekstowych — ang. *HyperText Transfer Protocol*).

9.1. Poczta elektroniczna

System pocztowy składa się z wielu współpracujących ze sobą komponentów. Czytając dokumentację dotyczącą poczty elektronicznej spotykamy się często z takimi ich nazwami jak *agent użytkownika* (*MUA* — ang. *Message User Agent*) i *agent transmisji* (*MTA* — ang. *Message Transmission Agent*), czy rzadszymi *MSA* (ang. *Mail Submission Agent*) i *MDA* (ang.

[1] W języku C można je zapisać jako `'\r'` i `'\n'`.

Mail Delivery Agent)[2]. Wykonują one różne zadania takie jak prezentacja wiadomości użytkownikowi i umożliwienie mu stworzenia takiej wiadomości (MUA), automatyczne przetwarzanie przychodzącej poczty i stosowanie technik ochrony przed *spamem* (MDA) czy przesyłanie poczty (np. MTA). Terminy te są związane raczej z wykonywanymi zadaniami niż z konkretnymi programami. Pojedyncza aplikacja może spełniać więcej niż jedną z tych funkcji.

W dalszej części rozdziału zajmować się będziemy programowaniem tych komponentów, które są bezpośrednio związane z przesyłaniem poczty. Opiszemy używane protokoły warstwy zastosowań oraz format wiadomości pocztowej i adresu e-mail. Wszystkie aspekty związane z programowaniem poczty elektronicznej zawarte są w książce [62]

Pierwszymi podstawowymi dokumentami opisującymi pocztę elektroniczną w postaci istniejącej obecnie są [47] opisujący sposób przesyłania wiadomości i [12] opisujący format wiadomości. Obecnie dokumenty te zastąpione są przez aktualniejsze [29] i [53].

9.1.1. Adres e-mail

Adres określający odbiorcę wiadomości składa się z dwóch części oddzielonych znakiem „`@`": części lokalnej i części określającej domenę, np. `jan.kowalski@example.org`.

Część domenowa używana jest do określenia hosta, do którego trzeba dostarczyć pocztę. W tym celu wysyłane jest do DNS zapytanie o rekord MX związany z tą domeną[3].

Oto przykład takiego zapytania wykonanego za pomocą programu `dig` dla domeny `hektor.umcs.lublin.pl`:

```
$ dig hektor.umcs.lublin.pl MX
;; ANSWER SECTION:
hektor.umcs.lublin.pl.  14400   IN    MX    1 hektor.umcs.lublin.pl.
hektor.umcs.lublin.pl.  14400   IN    MX    5 westa.umcs.lublin.pl.

;; ADDITIONAL SECTION:
hektor.umcs.lublin.pl.  14400   IN    A     212.182.57.178
westa.umcs.lublin.pl.   14400   IN    A     212.182.74.70
```

Liczby przed nazwami domen w zwróconych rekordach MX oznaczają priorytet — adres z mniejszą liczbą brany jest pod uwagę najpierw. Dodatkowe rekordy typu A wskazują od razu adresy IP odpowiednich hostów.

---

[2] Terminologia ta stosowana jest m.in. w dokumentach RFC i opisana jest w dokumencie [11].

[3] Dokładna procedura związana z uzyskaniem potrzebnego adresu opisana jest w sekcji 5.1 dokumentu [29].

Część lokalna interpretowana jest tylko przez hosta określonego przez część domenową. Nie może uczestniczyć w podejmowaniu decyzji gdzie przesłać pocztę (patrz [29] — 2.3.11). Najczęściej oznacza identyfikator użytkownika danego serwera, ale może być też związana z pewnymi usługami udostępnianymi przez serwer.

Małe i duże litery są rozróżniane w części lokalnej[4] i nierozróżniane w części domenowej.

Adresy pocztowe użyte w polach nagłówka wiadomości (patrz podrozdział 9.1.2.1) mogą być rozszerzone o dodatkowy komentarz[5] używany wyłącznie przez oprogramowanie MUA — nie jest on wykorzystywany do określenia adresata wiadomości. Przykładami takiego adresu mogą być:

— `"Jan Kowalski" <jan.kowalski@example.org>`,
— `sekretariat@firma.com (Sekretariat Firmy)`.

9.1.2. Format wiadomości pocztowej

Wiadomość pocztowa składa się z *koperty* związanej ściśle z protokołem SMTP i opisanej przez nas w podrozdziale 9.1.3.1 oraz dwóch części, którymi zajmiemy się tutaj: *nagłówka* i *ciała*.

Format nagłówka i ciała wiadomości e-mail został pierwotnie zdefiniowany w dokumencie [12]. Dokument ten został później zastąpiony przez [50], a następnie przez [53], ale nazwa *rfc822* jest w dalszym ciągu używana do określania tego typu wiadomości.

Ciało wiadomości jest oddzielone od nagłówka pojedynczym pustym wierszem (sekwencją CRLF).

9.1.2.1. Nagłówek wiadomości

Nagłówek jest podzielony na pola. Pole składa się z nazwy, dwukropka i wartości. Całe pole zapisane jest wyłącznie za pomocą znaków ASCII. Sposób kodowania znaków spoza ASCII w wartościach pól nagłówka podany jest w [39][6]. Wielkość liter w nazwach pól nie ma znaczenia. Najważniejsze pola wymienione są poniżej.

— Pola związane z odbiorcą (lub odbiorcami) wiadomości to:
  — `To:` — adresat wiadomości,
  — `Cc:` (ang. *Carbon Copy* — kopia, kalka) — adresat kopii,
  — `Bcc:` (ang. *Blind Carbon Copy*) — ukryty adresat kopii (inni odbiorcy nie są o nim w żaden sposób informowani).

---

[4] Nie muszą być — jest to zależne od konkretnego hosta, ale oprogramowanie pocztowe w żadnym wypadku nie może tego zakładać i wielkości liter zmieniać.

[5] Format takiego adresu opisany jest w sekcji 3.2 dokumentu [53].

[6] Przykład takiego kodowania, to:

```
Subject: =?UTF-8?Q?=C5=BByczenia_=C5=9Bwi=C4=85teczne?=
```

— Pola związane z nadawcą wiadomości, to:
  — `From:` — twórca wiadomości,
  — `Sender:` — adres osoby odpowiedzialnej za wysłanie wiadomości,
  — `Reply-To:` — do kogo ma być adresowana odpowiedź; służy programom pocztowym do automatycznego wypełniania pola odbiorcy podczas tworzenia odpowiedzi.
— Innymi ważnymi polami są:
  — `Date:` — obowiązkowe pole informujące o czasie wysłania wiadomości,
  — `Subject:` — temat wiadomości określony przez nadawcę[7],
  — `Message-Id:` — jednoznaczny identyfikator wiadomości,
  — `In-Reply-To:` — w odpowiedziach oznacza wiadomość, której dotyczy wiadomość,
  — `Received:` — określa drogę, którą przeszła wiadomość; każdy MTA po drodze dodaje od siebie jedno takie pole.

Pola zaczynające się od „`X-`" są polami niestandardowymi. Mogą ich używać różne aplikacje we właściwy dla siebie sposób bez ryzyka, że nowy standard zdefiniuje kiedyś pole o takiej nazwie.

Długie wartości pól mogą być podzielone na wiele wierszy w sposób przedstawionych np. na listingu 9.1 na stronie 114.

Minimalny zestaw pól nagłówka, to `Date:` i `From:`[8].

9.1.2.2. Ciało wiadomości i MIME

Dokument [12] stwierdzał jedynie, że ciało wiadomości podzielone jest na wiersze zawierające znaki 7-bitowego kodu ASCII. Nie nadawał poza tym ciału żadnej określonej struktury.

Mechanizmem nadającym ciału wiadomości strukturę jest *MIME* (ang. *Multipurpose Internet Mail Extensions*). Podstawowymi dokumentami opisującymi go są [18] i [19].

Oprócz nadania ciału wiadomości struktury MIME określa reguły kodowania wiadomości niedających się zapisać w ASCII.

Standard ten wprowadza do nagłówka wiadomości dodatkowe pola [18]. Obecność pola `MIME-Version:` (ma zawsze wartość `1.0`) oznacza, że wiadomość jest w formacie MIME. Pozostałe pola mogą występować również jako pola nagłówków poszczególnych części wiadomości MIME. Są one wymienione poniżej.

— `Content-Type:` określa typ, podtyp i format wiadomości np. `text/plain; charset=windows-1250;`. Typy opisane są w dokumencie

[7] Nie jest wymagany przez [53], ale netykieta wymaga użycia go.
[8] Może dziwić tutaj brak pola odbiorcy `To:`, ale nie oznacza to, że odbiorca może nie być znany — jest zawsze określony za pomocą koperty (patrz podrozdział 9.1.3.1).

[19]. Wszystkie zarejestrowane typy danych dostępne są online pod adresem `http://www.iana.org/assignments/media-types/`.

- `Content-Transfer-Encoding:` służy głównie do określenia sposobu radzenia sobie z danymi innymi niż czysty tekst ASCII — danymi binarnymi, tekstami z różnymi znakami narodowymi. Przykładowymi wartościami są `quoted-printable` (głównie do danych zawierających niewiele znaków spoza ASCII, np. tekstów w różnych językach) czy `base64` (głównie do danych binarnych).
- Pole `Content-Disposition:` zostało wprowadzone w dokumencie [59] (nie jest związane ze strukturą wiadomości). Określa sposób traktowania danej części wiadomości. Pole to może mieć wartość `attachment` — zawartość traktowana jest jako załącznik lub `inline` bądź — zawartość wyświetlana razem z całą wiadomością.

Pole `Content-Type` może służyć do wprowadzenia do wiadomości dodatkowej struktury. Zadanym typem pola musi wtedy być `multipart`. Przykładowymi wartościami podtypu mogą być wtedy np.:

- `alternative` — alternatywne wersje tej samej zawartości, np. wiadomość w formacie HTML i ta sama wiadomość czystym tekstem dla programów nieobsługujących HTML;
- `mixed` — części mają niezależną od siebie zawartość.

Poszczególne części wiadomości oddzielone są od siebie ciągiem znaków określonym przez parametr `boundary` pola `Content-Type` i zaczynają się od pól MIME. Każda z części wiadomości może w ten sam sposób wprowadzać wewnętrzną strukturę. Na listingu 9.1 zaprezentowany jest przykład takiej wiadomości.

Listing 9.1. Przykładowa wiadomość w formacie MIME

```
MIME-Version: 1.0
Received: by 10.152.29.98 with HTTP;
      Wed, 22 Feb 2012 01:19:31 -0800 (PST)
Date: Wed, 22 Feb 2012 10:19:31 +0100
Delivered-To: jan.kowalski@przykladowa.domena.com
Message-ID: <b23492849870889@mail.przykladowa.dom...
Subject: =?ISO-8859-2?Q?Wa=BFne_informacje?=
From: =?ISO-8859-2?Q?Micha=B3_Klisowski?=
      <michal.klisowski@przykladowa.domena.com>
To: =?ISO-8859-2?Q?Iksi=F1ski?=
      <iksinski@inna.domena.lublin.pl>
Cc: "Jan Kowalski"
      <janek@przykladowa.domena.com>
Content-Type: multipart/mixed;
      boundary=e0cb4efe2e9869a03f04b98b5fb8
```

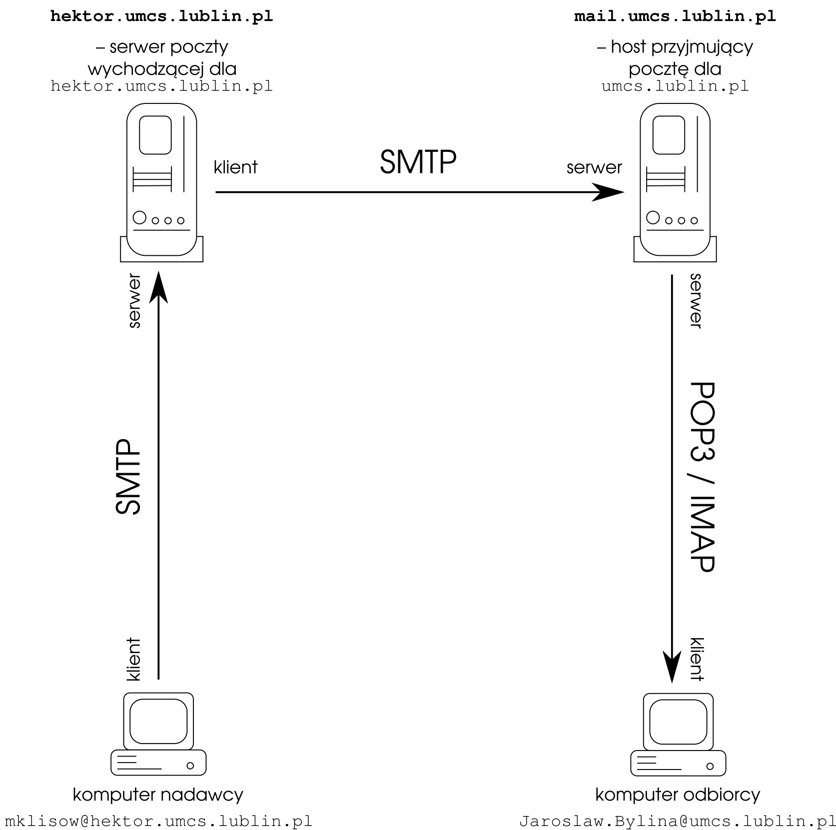
```
16

18 --e0cb4efe2e9869a03f04b98b5fb8
   Content-Type: multipart/alternative;
20       boundary=e0cb4efe2e9869a03704b98b5fb6

22 --e0cb4efe2e9869a03704b98b5fb6
   Content-Type: text/plain; charset=ISO-8859-2
24 Content-Transfer-Encoding: quoted-printable

26 W za=B3=B1czniku s=B1 *wa=BFne* informacje.
   Micha=B3
28
   --e0cb4efe2e9869a03704b98b5fb6
30 Content-Type: text/html; charset=ISO-8859-2
   Content-Transfer-Encoding: quoted-printable
32
   W za=B3=B1czniku s=B1 <b>wa=BFne</b> informacje.
34 <br>Micha=B3<br>

36 --e0cb4efe2e9869a03704b98b5fb6--
   --e0cb4efe2e9869a03f04b98b5fb8
38 Content-Type: text/plain; charset=windows-1250;
         name="=?ISO-8859-2?B?d2G/bmUudHh0?="
40 Content-Disposition: attachment;
         filename="=?ISO-8859-2?B?d2G/bmUudHh0?="
42 Content-Transfer-Encoding: base64
   X-Attachment-Id: f_gyy8z6mt0
44
   WmG/87PmIGfqnGy5IGphn/Eu
46 --e0cb4efe2e9869a03f04b98b5fb8--
```

9.1.3. Protokoły

Wspólnym zadaniem protokołów pocztowych jest dostarczanie wiadomości pocztowych od nadawcy do odbiorcy. Podstawowym protokołem służącym do przesyłania wiadomości jest *SMTP* opisany w [29]. Klient SMTP po nawiązaniu połączenia z serwerem przesyła mu wiadomość. W sytuacji gdy serwer jest odbiorcą wiadomości problemem mogłoby być to, że komputer osoby chcącej odbierać pocztę musiałby mieć cały czas możliwość akceptowania połączeń TCP od komputerów nadawców wiadomości. Taka sytuacja nie zawsze ma miejsce — domowy komputer użytkownika poczty może nie mieć stałego dostępu do Internetu, adres tego komputera może nie być publicznie

dostępny (np. ze względu na NAT — patrz podrozdział 2.2.1.2) albo może po prostu być wyłączony.

Żeby rozwiązać te problemy, wiadomości nie są wysyłane bezpośrednio do komputera odbiorcy, tylko do specjalnego serwera pocztowego, a odbiorca może odebrać je w każdej chwili łącząc się z nim przy użyciu protokołów *POP3* bądź *IMAP*. Nadawca zwykle nie wysyła wiadomości bezpośrednio do serwera obsługującego skrzynkę pocztową odbiorcy. Wysyła wiadomość do serwera odpowiedzialnego za odbieranie od niego poczty, a dopiero ten serwer wysyła wiadomość do serwera, z którego odbiorca będzie mógł ją odebrać. Umiejscowienie protokołów pocztowych w kolejnych etapach transmisji przedstawia rysunek 9.1.



Rysunek 9.1. Droga wiadomości pocztowej przez maszyny i protokoły

9.1.3.1. SMTP

Klient chcący przesłać pocztę za pomocą protokołu SMTP musi nawiązać połączenie TCP (standardowym portem usługi SMTP jest 25). Serwer od-

powiada powitaniem (kodem 220), po czym następuje seria komend klienta i odpowiedzi serwera.

Zadaniem protokołu SMTP jest przesłanie wiadomość na podstawie jej koperty.

Koperta tworzona jest zazwyczaj w oparciu o pola dotyczące nadawcy i odbiorcy z nagłówka wiadomości[9]. Ewentualne pole `Bcc:` powinno zostać podczas tworzenia koperty usunięte z nagłówka[10].

Koperta określona jest za pomocą dwóch komend:

— `MAIL` określa adres nadawcy wiadomości. Tylko jedna taka komenda może wystąpić dla danej koperty.
— `RCPT` określa odbiorcę. Jedna komenda jest wykonywana dla każdego odbiorcy wiadomości.

Inne wybrane komendy SMTP to:

— `HELO` lub `EHLO` to pierwsze polecenie wysyłane serwerowi. Argumentem powinna być pełna nazwa domenowa klienta. W oryginalnej definicji SMTP występowała jedynie komenda `HELO`. `EHLO` jest skrótem od *Extended HELLO* i obecnie jest zdecydowanie częściej używana — umożliwia stosowanie różnych rozszerzeń SMTP (w odpowiedzi na tę komendę serwer wysyła listę wspieranych rozszerzeń). Protokół SMTP z rozszerzeniami określa się często nazwą ESMTP (Extended Simple Mail Transfer Protocol). Jeżeli serwer nie rozumie polecenia `EHLO`, to klient powinien wysłać zwykłe `HELO`.
— `DATA` oznacza rozpoczęcie podawania wiadomości. Koniec oznaczany jest przez wiersz zawierający samą kropkę.
— `QUIT` oznacza zakończenie połączenia przez klienta.

Przykładowa sesja wysyłająca wiadomość do jednego odbiorcy i dwóch odbiorców ukrytej kopii przedstawiona jest na poniższym listingu (Wiersze wysyłane przez klienta zaznaczone są znakiem „>”).

```
  220 smtp.example.com ESMTP
> HELO [192.168.1.101]
  250 Hello [192.168.1.101]
> MAIL FROM:<jan.kowalski@example.org>
  250 Ok
> RCPT TO:<maciek@host1.com>
  250 Ok
```

[9] Dla SMTP nie ma znaczenia zawartość przesyłanej wiadomości — jeżeli dane w nagłówku nie będą spójne z danymi w kopercie, to wiadomość nie będzie dostarczona zgodnie z nagłówkiem. Programy pocztowe powinny zadbać o tę spójność.

[10] Proces generowania koperty na podstawie nagłówka opisany jest w dodatku B dokumentu [29].

```
> RCPT TO:<tata.macka@host2.org>
  250 Ok
> RCPT TO:<mama.macka@host2.org>
  250 Ok
> DATA
  354 End data with "." on a line by itself
> From: "Jan Kowalski" <jan.kowalski@example.org>
> To: "Maciek" <maciek@host1.com>
> Date: Date: Wed, 22 Feb 2012 16:37:22 +0100
> Subject: Test
>
> 1. wiersz
> 2. wiersz
> .
  250 Ok
> QUIT
  221 Bye
```

Znaczenie różnych kodów błędów w odpowiedziach serwera można znaleźć w sekcji 4.2 dokumentu [29]. W odpowiedziach serwera praktyczne znaczenie dla klienta ma jedynie numer rozpoczynający odpowiedź. Reszta może jedynie służyć do celów diagnostycznych. Wielkość liter w komendach nie ma znaczenia. W powyższym przykładzie wszystkie żądania serwera kończą się odpowiedziami pozytywnymi.

9.1.3.2. Protokoły odbioru wiadomości: POP3 i IMAP

W tym podrozdziale omówione zostaną dwa najpopularniejsze protokoły służące do odbierania poczty elektronicznej — POP3 i IMAP. Są to protokoły obsługiwane obecnie przez większość tworzonych klientów poczty.

Pierwszym — starszym i dużo prostszym — jest POP3. Zawiera on tylko minimalny zestaw operacji służących do odbierania poczty. Bardziej skomplikowane operacje takie jak wyszukiwanie danych w wiadomościach pocztowych, czy przeglądanie struktury tych wiadomości muszą być wykonane na komputerze klienta, już po ściągnięciu poczty.

Drugim — nowszym, bardziej skomplikowanym, ale mającym przez to dużo większe możliwości — jest IMAP. IMAP pozwala na wykonanie zdalnie na serwerze wielu czynności, które w przypadku POP3 były możliwe tylko po ściągnięciu poczty.

POP3. Protokół POP3 [40] jest prostym protokołem pozwalającym na odbiór poczty elektronicznej. Standardowym portem usługi POP3 jest 110. Po nawiązaniu przez klienta połączenia TCP serwer wysyła powitanie, a następnie klient i serwer wymieniają komendy i odpowiedzi. Odpowiedzi zaczynają się od `+OK` — odpowiedź pozytywna lub `-ERR` — odpowiedź negatywna.

Odpowiedzi na niektóre komendy są wielowierszowe. W takim przypadku ostatni wiersz odpowiedzi zawiera pojedynczą kropkę (i sekwencję CRLF).

Wielkość liter w komendach nie ma znaczenia. `ERR` i `OK` w odpowiedziach serwera muszą być zapisane wielkimi literami.

Działanie protokołu opisywane jest zazwyczaj za pomocą kolejnych stanów, w których znajduje się sesja. Na początku serwer wysyła przywitanie np.:

```
  +OK Dovecot ready.
```

Następnie sesja przechodzi w *stan autoryzacji* (ang. *Authorization State*). W tym momencie serwer oczekuje na użycie przez klienta jednego z mechanizmów autoryzacji udostępnianych przez serwer. Najprostszym sposobem jest użycie komend `user` i `pass`:

```
> user barnaba
  +OK
> pass bardzopoufne
  +OK Logged in.
```

Sesja przechodzi w stan transakcji (ang. *Transaction State*):

```
> stat
  +OK 2 4916
> list
  +OK 2 messages:
  1 2438
  2 2478
  .
> retr 1
  +OK 2438 octets
  Date: Fri, 24 Feb 2012 04:27:38 +0100
  Subject: Greetings
  From: john@example.com
  To: barnaba@firma.przykladowa.pl

  Greetings from John!
  .
> dele 1
  +OK Marked to be deleted.
> list 2
  +OK 2 2478
> retr 2
  +OK 2478 octets
  Date: Fri, 24 Feb 2012 04:27:53 +0100
  Subject: Pozdrowienia
  From: marysia@example.pl
  To: barnaba@firma.przykladowa.pl
```

```
  Pozdrawiam
  Marysia
  .
> dele 2
  +OK Marked to be deleted.
```

Komenda `stat` wyświetla informację o liczbie i łącznym rozmiarze wiadomości w skrzynce. Komenda `list` — rozmiar każdej wiadomości osobno. Komenda `retr` służy do pobrania całej wiadomości, a `dele` do zaznaczenia wiadomości jako przeznaczonej do usunięcia. Wszystkie zaznaczenia do usunięcia są cofane komendą `reset`. Jak widać POP3 nie rozpoznaje struktury wiadomości — polecenie `retr` umożliwia tylko ściągnięcie całej wiadomości. Jest jeszcze polecenie `top`, ale pozwala ono tylko określić ile wierszy danej wiadomości chcemy ściągnąć oprócz nagłówka. Poza tym jest to polecenie opcjonalne (serwer nie musi go implementować).

Kolejną komendą jest:

```
> quit
```

Po wykonaniu tej komendy w stanie transakcji sesja przechodzi w *stan aktualizacji* (ang. *Update State*). Dopiero teraz są ewentualnie usuwane wiadomości i sesja kończy się.

```
> +OK Logging out, messages deleted.
```

Wiersze wysyłane przez klienta w powyższym przykładzie zaznaczone są znakiem „>".

IMAP. Alternatywą dla POP3 jest IMAP [10]. Jest to protokół dużo bardziej rozbudowany i elastyczniejszy niż POP3. Założeniem tego protokołu jest — w przeciwieństwie do POP3 — przechowywanie całej poczty na serwerze bez konieczności jej ściągania na dysk lokalny oraz udostępnianie zdalnych operacji, które normalnie kojarzone są z operacjami na lokalnych skrzynkach pocztowych. Są to takie operacje jak zarządzanie folderami, przeszukiwanie wiadomości, czy przeglądanie ich struktury. Oczywiście ściągnięcie całej poczty na dysk lokalny i usunięcie jej z serwera, tak jak jest to zazwyczaj robione w przypadku jest POP3, też jest możliwe.

Cechą charakterystyczną protokołu IMAP jest to, że każde zapytanie poprzedzone jest napisem (tagiem) jednoznacznie wyróżniającym go od innych zapytań. Odpowiedź na pytanie będzie zawierała ten sam tag co pytanie. Serwer SMTP może w każdej chwili przesłać także informacje niebędące odpowiedzą na właśnie wysłane żądanie — bez tagu. Są one oznaczane znakiem `*`. Cechą charakterystyczną protokołu IMAP jest właśnie to, że klient powinien być w każdej chwili gotowy do przyjęcia danych od serwera. Jest

to związane z umożliwieniem wielu równoczesnych połączeń z jedną skrzynką. O zmianach wprowadzonych przez jedno połączenie inne połączenia są informowane.

W opisie protokołu IMAP — tak samo jak w opisie POP3 — używa się pojęcia stanu. Klient chcąc połączyć się z serwerem nawiązuje połączenie TCP (domyślny port to 143). Serwer wysyła pozdrowienie:

```
  * OK Dovecot ready.
```

Po pozdrowieniu sesja przechodzi w *stan nieuwierzytelniony* (ang. *Nonauthenticated State*). Polecenie `CAPABILITY` jest prośbą o zwrócenie informacji o mechanizmach wspieranych przez serwer (rozszerzeniach, sposobach logowania). Jest to jedna z komend IMAP dostępna we wszystkich stanach.

```
> 0001 CAPABILITY
* CAPABILITY IMAP4rev1 SASL-IR SORT THREAD=REFERENCES ...
  0001 OK Capability completed.
```

Po zalogowaniu się klienta (np. poleceniem `LOGIN`) sesja przechodzi w *stan uwierzytelniony* (ang. *Authenticated State*):

```
> 0002 LOGIN alibaba sezamieotworzsie
  0002 OK Logged in.
```

W tym stanie klient powinien wybrać poleceniem `SELECT` skrzynkę pocztową, na której będzie pracował. Innym możliwym poleceniem jest `EXAMINE` — otwarcie tylko do odczytu.

```
> 0003 SELECT Inbox
  * FLAGS (\Answered \Flagged \Deleted \Seen \Draft)
  * OK [PERMANENTFLAGS (\*)] Flags permitted.
  * 3 EXISTS
  * 1 RECENT
  * OK [UIDVALIDITY 1309532744] UIDs valid
  * OK [UIDNEXT 15] Predicted next UID
  0003 OK [READ-WRITE] Select completed.
```

W odpowiedzi serwer przesyła m.in. kilka obowiązkowych informacji: `EXISTS` — ile jest wiadomości w skrzynce, `RECENT` — liczba wiadomości nie widzianych wcześniej oraz `FLAGS` i `PERMANENTFLAGS` — jakie *flagi* są dostępne w danej skrzynce i które z nich klient może zmieniać. Flagi są pojęciem związanym z protokołem `IMAP` nie spotykanym w protokołach POP3 i SMTP. Każda wiadomość może zostać „oznakowana” za pomocą jednej lub wielu flag. Przykładowymi flagami są `\Seen` (wiadomość przeczytana), `\Answered` („odpowiedziana”), `\Deleted` (zaznaczona do skasowania) czy `\Recent` (obecna sesja jest pierwszą, podczas której użytkownik widzi te wiadomości). Po otwarciu skrzynki sesja jest w *stanie wybranym* (ang. *Selected State*). W tym momencie można wykonywać operacje na wiadomościach

danej skrzynki, np. FETCH (pobieranie danych), czy SEARCH (wyszukiwanie danych). W poniższym przykładzie pobierane są tematy wszystkich wiadomości ze skrzynki;

```
> 0004 FETCH 1:3 BODY[HEADER.FIELDS (SUBJECT)]
  * 1 FETCH (BODY[HEADER.FIELDS (SUBJECT)] {35}
  Subject: Pewien trudny problem

  )
  * 2 FETCH (BODY[HEADER.FIELDS (SUBJECT)] {25}
  Subject: Praca domowa

  )
  * 3 FETCH (BODY[HEADER.FIELDS (SUBJECT)] {16}
  Subject: PKS

  )
  0004 OK Fetch completed.
```

W poniższym przykładzie zwracane są numery wszystkich wiadomości zawierających w swoim ciele słowo problem.

```
> 0005 search 1:3 BODY "problem"
  * SEARCH 2 3
  0005 OK Search completed.
```

Na tych przykładach widać, że w przeciwieństwie do POP3 IMAP potrafi rozpoznać strukturę wiadomości. Po zakończeniu operacji na serwerze klient wydaje polecenie LOGOUT:

```
> 0006 LOGOUT
  * BYE Logging out
  0006 OK Logout completed.
```

Polecenie to powoduje chwilowe przejście w *stan wylogowania* (ang. *Logout State*) i zakończenie sesji.

W przeciwieństwie do scenariuszy przedstawionych przy okazji omawiania protokołów SMTP i POP3 nie można powiedzieć, że przedstawiony scenariusz jest typową sekwencją poleceń — w przypadku protokołu IMAP sposobów jego użycia (i dostępnych poleceń) jest bardzo dużo i trudno któryś z nich nazwać typowym. Dostępnych jest mnóstwo innych, nieprzedstawionych powyżej poleceń oraz parametrów poleceń.

9.1.4. Webmail

Obecnie coraz popularniejsze jest używanie serwisów WWW do obsługi poczty (*Webmail*). Nie trzeba wtedy mieć zainstalowanych na komputerze

klientów protokołów pocztowych opisanych powyżej — wystarczy przeglądarka internetowa, czyli klient protokołu HTTP opisanego w podrozdziale 9.2. Nie oznacza to jednak, że opisane protokoły nie biorą udziału w przesyłaniu poczty. Po prostu w tej sytuacji przeglądarka internetowa wraz z oprogramowaniem serwera WWW spełnia rolę MUA[11].

9.1.5. Bezpieczeństwo poczty

9.1.5.1. Szyfrowanie

Używając poczty elektronicznej należy zdawać sobie sprawę z kilku rzeczy dotyczących jej bezpieczeństwa. Pokazaliśmy w tym rozdziale standardowe sposoby uwierzytelniania w protokołach POP3 i IMAP. W SMTP uwierzytelnianie dostępne jest tylko jako rozszerzenie (ESMTP). Podstawowe metody autoryzacji SMTP jak i metody przedstawione dla POP3 i IMAP przekazują loginy i hasła czystym tekstem. Każdy kto ma fizyczny dostęp do sprzętu transmitującego nasze dane ma możliwość łatwego odczytania ich.

Stosowanie tych metod jest bezpieczne tylko wtedy, gdy stosowane hasła są jednorazowe lub cała komunikacja jest szyfrowana[12]. Wszystkie powyższe protokoły mają przydzielony standardowy port, na którym transmisja odbywa się zabezpieczonym kanałem komunikacyjnym. Dla SMTP jest to port 465, dla POP3 — 995, a dla IMAP — 993. Poza tym każdy z tych protokołów ma rozszerzenie *STARTTLS* umożliwiające rozpoczęcie szyfrowania transmisji w ramach nieszyfrowanej sesji już rozpoczętej na standardowym porcie.

9.1.5.2. Fałszowanie poczty (Fake-mail)

Trzeba zdawać sobie sprawę także z tego, że uwierzytelnianie SMTP nie pozwala na dostęp do serwera SMTP niepowołanym osobom, ale w żaden sposób nie zabezpiecza przed tworzeniem wiadomości pocztowych z fałszywymi danymi w kopercie lub nagłówku wiadomości. Jest to analogiczna sytuacja do standardowej poczty — każdy może wysłać nam list z wpisanymi dowolnymi danymi w polu nadawcy. Trudniej będzie za to odebrać pocztę adresowaną do nas bez posiadania kluczyka do naszej skrzynki pocztowej.

---

[11] W tej sytuacji użycie POP3 lub IMAP może nie być potrzebne, jeżeli np. serwer przyjmujący dla nas pocztę przez SMTP jest fizycznie na tej samej maszynie co serwer WWW.

[12] Za pomocą TLS — ang. *Transport Layer Security*.

9.2. WWW

WWW — ang. World Wide Web jest chyba najpopularniejszą usługą w dzisiejszym Internecie — na tyle popularną, że obecnie usługi sieciowe wymagające wcześniej osobnych programów, są często realizowane za jej pośrednictwem. Przykładem może być Webmail opisany w podrozdziale 9.1.4. Co więcej aplikacje bazujące na WWW i przeglądarkach internetowych są obecnie często wykorzystywane do celów, które tradycyjnie realizowane były za pomocą programów w ogóle nie wymagających dostępu do sieci, jak np. różne pakiety biurowe. Powstają również aplikacje WWW mające z założenia zastąpić tradycyjny system operacyjny z zainstalowanymi programami użytkowymi[13]. Jedynym wymaganiem dotyczącym lokalnego systemu jest wówczas przeglądarka internetowa mogąca obsłużyć taki program.

Podstawowym protokołem warstwy aplikacji, na którym bazuje WWW jest *HTTP*.

9.2.1. URL

Zadaniem serwera HTTP jest udostępnianie klientom (np. przeglądarkom internetowym) zasobów. Położenie zasobów określane jest za pomocą *jednolitego identyfikatora zasobów* (ang. *URL — Unified Resource Locator*).

Składnia URL. Składnię URL[14] dla protokołu HTTP można opisać jako `http://<host>:<port>/<ścieżka>?<zapytanie>#<fragment>`.

Części `<host>` i `port` oznaczają adres serwera udostępniającego zasób. `<host>` oznacza adres domenowy lub IP, a `<port>` — port, z którym należy nawiązać połączenie. W razie braku części `:<port>` przyjmowany jest domyślny dla protokołu HTTP port 80.

Część `/<ścieżka>?<zapytanie>` jest w niezmienionej postaci przesyłana w zapytaniu i jej interpretacja należy wyłącznie do serwera. Część `<ścieżka>` określa zazwyczaj położenie zasobu na serwerze i jest często interpretowane przez serwer jako odniesienie do konkretnego pliku bądź katalogu w systemie plików serwera. Na przykład `/doc/index.html` może być przez serwer zinterpretowane jako lokalny plik `/var/www/doc/index.html`. Część `<zapytanie>` ma sens tylko dla zasobów generowanych dynamicznie — tzn. serwer po otrzymaniu żądania uruchamia odpowiedni program (skrypt) generujący odpowiedź — i określa parametry działania tego skryptu[15]. Jeżeli część `/<ścieżka>?<zapytanie>` w ogóle nie występuje, to przyjmuje się że

[13] Jest to tzw. *WebOS* (ang. *Web operating system*).
[14] Dokładny opis składni znajduje się w sekcji 3.3 dokumentu [6] i w dokumencie [5].
[15] Zapytanie ma często postać `parametr1=wartość1¶metr2=wartość2...`

składa się ona z pojedynczego znaku / (i w takiej postaci jest przesyłana serwerowi).

Część #<fragment> nie ma dla protokołu HTTP żadnego znaczenia. Jest ona interpretowana przez klienta (przeglądarkę) i jeżeli występuje, to określa pozycję w żądanym dokumencie (zwykle dotyczy to strony HTML). Nie jest w ogóle brana pod uwagę przy wysyłaniu do serwera żądania HTTP.

Przykładem URL-a może być `http://matrix.umcs.lublin.pl/~mklisow/slownik.php?slowo=network#wytlumaczenie`. Kolejnymi częściami opisanymi przez nas powyżej są wówczas:

— host: `matrix.umcs.lublin.pl`,
— port jest pominięty,
— ścieżka: `~mklisow/slownik.php`,
— zapytanie: `slowo=network`,
— fragment: `wytlumaczenie`.

Niektóre znaki nie mogą z różnych powodów występować w URL-u. Np. spacja w URL-u mogłaby zostać zinterpretowana jako element oddzielający go od innych danych. Niektóre znaki mają też w URL-u specjalne znaczenie (np. #, ?, / czy =). Jeżeli chcemy umieścić taki znak w URL-u lub pozbawić go specjalnego znaczenia, to musimy zapisać go w systemie szesnastkowym (jako wartość jego kodu) i poprzedzić znakiem %. Przykładem takiego zakodowanego URL-a może być: `http://host/test?pole=ze%20spacj%C4%85`. Taki sposób kodowania (ang. *URL encoding*) używany jest także do przekształcania danych pochodzące z formularzy na stronach WWW (które też mogą być umieszczone w URL-u). W przypadku formularzy spacja zamieniana jest na znak + zamiast na %20.

9.2.2. Protokół HTTP

Pierwszą udokumentowaną wersją protokołu HTTP była wersja znana obecnie jako HTTP/0.9. Był to wyjątkowo prosty protokół, którego zadaniem było wyłącznie przesyłanie stron HTML. Cały jego opis mieścił się na kilku stronach (jest dostępny tutaj [3]). Wersją zdecydowanie bardziej rozbudowaną i dużo dokładniej udokumentowaną jest wersja HTTP/1.0 opisana w [4]. Oficjalnym standardem (protokół HTTP/1.0 nigdy takim nie był) jest wersja HTTP/1.1 opisana w [16]. Omówimy dalej protokół HTTP skupiając się tej właśnie wersji 1.1. W niektórych miejscach opiszemy różnice między nim a HTTP/1.0, który w dalszym ciągu jest jeszcze w użyciu[16].

[16] Protokołem HTTP/0.9 nie będziemy się w ogóle zajmować. W dokumencie [16] wspomina się jedynie, że komercyjne serwery HTTP/1.1 powinny obsługiwać żądania HTTP/1.0 i HTTP/0.9, a klienty — odpowiedzi w tych protokołach.

Przeanalizujemy działanie protokołu HTTP bazując na poniższym listingu pokazującym przykładowe żądanie i odpowiedź HTTP (Wiersze żądania zaznaczone są znakiem „>”).

Listing 9.2. Przykładowe żądanie i odpowiedź HTTP

```
   > GET /przyk%C5%82ady/test.html HTTP/1.1
 2 > Host: matrix.umcs.lublin.pl
   > User-Agent: Mozilla/5.0 (Windows NT 5.1;
 4 >       rv:10.0.2) Gecko/20100101 Firefox/10.0.2
   > Accept: text/html,application/xhtml+xml,
 6 >       application/xml;q=0.9,*/*;q=0.8
   > Accept-Language: pl,en-us;q=0.7,en;q=0.3
 8 > Accept-Encoding: gzip, deflate
   > Connection: keep-alive
10 > If-Modified-Since: Thu, 23 Feb 2012 18:48:39 GMT
   > If-None-Match: "d20004-c0-4b9a617a14fc0"
12 >
   HTTP/1.1 200 OK
14 Date: Thu, 23 Feb 2012 18:49:18 GMT
   Server: Apache/2.3.15-dev (Unix)
16     mod_ssl/2.3.15-dev OpenSSL/1.0.0c
   Last-Modified: Thu, 23 Feb 2012 18:49:12 GMT
18 Etag: "d20004-c0-4b9a61998da00"
   Accept-Ranges: bytes
20 Vary: Accept-Encoding
   Content-Encoding: gzip
22 Content-Length: 163
   Keep-Alive: timeout=15, max=100
24 Connection: Keep-Alive
   Content-Type: text/html
26
   <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
28  "http://www.w3.org/TR/html4/strict.dtd">
   <html>
30 <head>
     <title>Strona testowa</title>
32 </head>
   <body>
34 <p>Strona testowa!</p>
   </body>
36 </html>
```

Klient nawiązuje z serwerem połączenie TCP. Adres hosta i numer portu określone są na podstawie danych z URL-a. Następnie następuje wymiana żądań i odpowiedzi. Pojedyncze żądanie dotyczy zwykle prośby o przesłanie

zasobu zadanego URL-em. Odpowiedź zwykle zawiera ten zasób lub zwraca inne użyteczne informacje.

W formacie wiadomości (żądaniu lub odpowiedzi) HTTP można wyróżnić trzy części — pierwszy wiersz (o składni różniącej się w przypadku żądania i odpowiedzi), *nagłówek* składający się *z pól nagłówkowych* (nazywanych też często po prostu nagłówkami) zgodny z formatem rfc822 [12] oraz oddzieloną od nagłówka pustym wierszem *część zasadniczą*. Części zasadniczej może, w zależności od typu żądania lub odpowiedzi, nie być. W powyższym przykładzie zapytanie nie ma części zasadniczej, a odpowiedź zwraca w części zasadniczej żądany zasób — stronę HTML.

Pierwszy wiersz żądania składa się z wybranej metody (`GET`), żądanego zasobu `/przyk%C5%82ady/test.html` i wersji protokołu `HTTP/1.1`. Metoda `GET` jest najczęściej stosowaną metodą. Służy do zwykłego pobrania danych z serwera (inne dwie powszechnie używane metody — `POST` i `HEAD` są omówione w dalszych podrozdziałach).

Pierwszy wiersz odpowiedzi składa się z określenia wersji protokołu oraz kodu odpowiedzi (i związaną z nią krótką frazą) — wartość kodu 200 (i fraza `OK`) oznacza odpowiedź serwera zwracającą żądany zasób.

Omówimy teraz niektóre z pól wiadomości HTTP oraz związane z nimi zagadnienia.

Pole `User-Agent` pozwala umieścić w żądaniu informacje o oprogramowaniu klienta. Umożliwia to np. dynamiczne dopasowanie wysyłanych treści do używanej przeglądarki[17]. Analogicznym polem w odpowiedzi jest pole `Server`.

9.2.2.1. Pole `Host` i serwery wirtualne

Pole `Host` (może występować tylko w żądaniu) jest jedynym obowiązkowym polem w żądaniu HTTP (w wersji 1.1, w poprzednich nie). Pole powinno zawierać nazwę hosta użytą w URL-u. Jest to przydatne w przypadku działania wielu *serwerów wirtualnych* korzystających z jednego adresu IP. Serwer dzięki temu wie, nazwa którego serwera została użyta.

9.2.2.2. Połączenia w HTTP

Podstawową różnicą między protokołem HTTP/1.1 a HTTP/1.0 jest obsługa połączeń. W HTTP/1.0 każdemu połączeniu odpowiadała jedna para żądanie — odpowiedź. W HTTP/1.1 domyślnie serwer nie zamyka połączenia po odesłaniu odpowiedzi, tylko czeka na dalsze żądania. Dla przykładu typowa strona WWW składa się z wielu plików (oprócz pliku HTML także obrazki, skrypty, arkusze stylów) i każdy z nich wymaga osobnego

[17] np. ze względu na zastosowanie różnych sztuczek w języku HTML związanych z obejściem różnic w interpretowaniu standardów przez przeglądarki.

wysłanego żądania. Pobranie ich wszystkich za pomocą jednego połączenia przyspiesza pobieranie strony i zmniejsza liczbę potrzebnych do zrealizowania tego segmentów TCP. Pole to w żądaniu oznacza zachowanie, o jakie prosi klient, a w odpowiedzi — informację o tym, jak serwer rzeczywiście się zachowa. Wartościami pola `Connection` mogą być `close` (zamknięcie połączenia po wysłaniu żądania — domyślne w starszych wersjach) lub domyślna w HTTP/1.1 wartość — `Keep-Alive` (czekanie na dalsze połączenia). W przypadku niezamykania połączenia sposobem na poinformowanie klienta o rozmiarze zasobu[18] jest pole `Content-Length` lub wartość `chunked` w polu `Transfer-Encoding`[19]. Ten drugi przypadek oznacza, że ciało (część zasadnicza) wiadomości zostanie wysłane w częściach, każda z informacją o jej rozmiarze. Pozwala to na rozpoczęcie wysyłania przez serwer przed zakończeniem generowania odpowiedzi (np. przez skrypt).

Utrzymywanie połączeń pozwala też na *przesyłanie potokowe* (ang. *pipelining*) — klient może wysyłać kilka żądań (ale nie żądań POST) bez oczekiwania na odpowiedź, a dopiero potem rozpocząć odbieranie odpowiedzi. Może to przyspieszyć pobieranie stron — po pierwsze kilka żądań może być nawet zebranych w jednym segmencie TCP, po drugie serwer po odesłaniu odpowiedzi nie musi czekać na przesłanie kolejnego żądania.

9.2.2.3. Zawartość części zasadniczej

Pole `Content-Type` określa typ[20] ciała wiadomości. Za pomocą parametru `charset` można określić też zestaw znaków dla typu wymagającego tego (przykładową wartością pola może być `text/html; charset=ISO-8859-2`). Pole to powinno być obecne w każdej wiadomości mającej część zasadniczą. Na podstawie tego pola klient może zdecydować o tym, co ma zrobić z otrzymanym zasobem[21].

Polami związanymi ze sposobem interpretowania ciała wiadomości są też np. wspomniane już pole `Transfer-Encoding` (i wartość `chunked`) określające przekształcenia zastosowane do ciała wiadomości w celu przetranspor-

[18] W przypadku, gdy serwer zamyka połączenie, klient rozpoznaje koniec zasobu po końcu połączenia.

[19] Mechanizm opisany w sekcji 3.6.1 dokumentu [16].

[20] Analogicznie do mechanizmu MIME.

[21] I tylko na jego podstawie. Niedopuszczalne jest decydowanie o tym np. na podstawie treści bądź rozszerzenia pliku umieszczonego w części ścieżkowej URL-a (jedynym wyjątkiem jest brak pola `Content-Type` w odpowiedzi.). Np. zasób dostępny przez URL kończący się rozszerzeniem `.html`, ale wysłany z typem `text/plain` nie powinien być potraktowany jako strona HTML, tylko jako zwykły tekst (przeglądarka nie powinna interpretować znaczników). Analogiczna sytuacja ma miejsce w przypadku coraz popularniejszego formatu XHTML wysyłanego jako `text/html` (a tak zazwyczaj jest) zamiast `application/xhtml+xml`[24]. Nie jest on wtedy interpretowany jako XML mimo posiadania deklaracji XML w treści.

towania go oraz `Content-Encoding` określające przekształcenie zastosowane do ciała wiadomości w celu np. kompresji[22]. Z `Content-Encoding` związane jest też pole `Accept-Encoding` (w naszym przypadku ma wartość `gzip, deflate`) wysyłane w żądaniach i określające, jakie kodowanie klient akceptuje. Domyślnym kodowaniem jest `identity` (brak przekształcenia) mogące się pojawić tylko jako wartość `Accept-Encoding`.

9.2.2.4. Przekierowania

Przydatnym mechanizmem HTTP są przekierowania. Pozwalają one serwerowi skierować klienta w inne miejsce w poszukiwaniu zasobu. Odpowiedź przekierowująca musi zawierać pole `Location` wskazujące nowy URL zasobu. Dwa podstawowe rodzaje przekierowań to:

— `301 Moved Permanently` oznaczające przeniesienie zasobu na stałe (kolejne żądania dotyczące zasobu powinny już być kierowane pod nowy URL);
— `302 Found` oznaczające tymczasowe przeniesienie zasobu pod wskazany URL (kolejne żądania nie powinny być kierowane pod nowy URL).

9.2.2.5. Pamięć podręczna (ang. *cache*)

Pobrane zasoby mogą być przechowywane w pamięci podręcznej (np. przez przeglądarkę). Lepsze wsparcie dla niej jest jedną z cech charakterystycznych dla wersji 1.1 protokołu HTTP. Polami wpierającymi mechanizm pamięci podręcznej są m.in. pola:

— `Date` określające bieżącą datę — obowiązkowe w HTTP/1.1;
— `Last-Modified` — czas ostatniej modyfikacji zasobu;
— `If-Modified-Since` — serwer wysyła zasób tylko, jeżeli został zmodyfikowany od zadanego czasu (w przeciwnym razie zwraca `304 Not Modified`;
— `If-Unmodified-Since` — serwer wysyła zasób tylko, jeżeli nie został zmodyfikowany od zadanego czasu (w przeciwnym razie zwraca `412 Precondition Failed`)[23].

9.2.2.6. Sesje w HTTP i ciasteczka

HTTP jest protokołem *bezstanowym*, co oznacza, że z punktu widzenia protokołu HTTP np. ciąg wizyt na kolejnych podstronach serwisu WWW

[22] Na listingu 9.2 wartość tego pola to `gzip` więc ciało wiadomości powinno mieć naprawdę inną postać niż przedstawiona na listingu — w rzeczywistości są tam dane binarne. Zdecydowaliśmy się na przedstawienie go po prostu w postaci czytelniejszej.

[23] Ma to sens np. w przypadku zastosowania razem z mechanizmem pobierania tylko części zasobu (pole `Range` w żądaniu i `Content-Range` w odpowiedzi). Jeżeli część zasobu została pobrana wcześniej, to pobranie kolejnej nie ma sensu, jeżeli w międzyczasie zasób się zmienił.

jest ciągiem zupełnie niezależnych od siebie par żądań i odpowiedzi. Jednak taka bezstanowość nie zawsze jest pożądana. Aplikacje oparte na WWW często muszą przechowywać informację związaną z użytkownikiem pomiędzy poszczególnymi żądaniami. Np. po zalogowaniu się na stronie i przejściu na inną stronę serwer musi zachować jakoś informację o tym, że jesteśmy zalogowani. Jednym ze sposobów zachowania informacji o kliencie jest zachowanie jego adresu IP. Ten sposób nie nadaje się jednak do wielu zastosowań — adres IP może nie być związany z pojedynczym komputerem (np. ze względu na NAT — patrz podrozdział 2.2.1.2), a już na pewno nie można za jego pomocą rozróżnić różnych użytkowników tego samego komputera. Inną metodą może być dodawanie informacji o użytkowniku (np. tzw. *identyfikatora sesji*) do każdego generowanego na stronie URL-a. Najczęściej używanym mechanizmem są *ciasteczka* (ang. cookies).

Mechanizm ciasteczek opisany jest w dokumencie [2]. Ciasteczka są krótkimi informacjami tekstowymi zapamiętywanymi (najczęściej zapisywanymi na dysku) przez klienta HTTP. Są to najczęściej informacje potrzebne do zidentyfikowania użytkownika. Mechanizm ciasteczek oparty jest o dwa pola nagłówkowe: `Set-Cookie` oznaczające przesłanie klientowi ciasteczka. W kolejnych żądaniach do hosta, który ustanowił ciasteczko używane będzie pole `Cookie` z zawartością ciasteczka — umożliwia to identyfikację użytkownika. W przypadku mechanizmu ciasteczek informacje potrzebne do zidentyfikowania klienta przechowywane są po jego stronie, w związku z czym użytkownik może bez problemu je usunąć, usuwając ciasteczka.

9.2.2.7. Uwierzytelnianie

HTTP oferuje podstawowe wbudowane w protokół mechanizmy umożliwiające uwierzytelnianie użytkownika opisane w [17]. Jednak dużo częściej spotykane jest uwierzytelnianie za pomocą loginu i hasła wpisywanych w zwykłe pola formularza HTML. Dane te są zwykle przesyłane na serwer za pomocą metody POST (patrz podrozdział 9.2.2.9). Informacja o tym, że użytkownik jest uwierzytelniony (zalogowany), przechowywana jest zwykle w ciasteczkach.

9.2.2.8. Kody odpowiedzi

W tym miejscu podamy informacje o najważniejszych grupach kodów odpowiedzi serwera.

— `2xx` — Żądanie zakończyło się sukcesem (np. `200 OK`, `206 Partial Content` — w przypadku żądania części zasobu).
— `3xx` — Przekierowanie (patrz podrozdział 9.2.2.4).
— `4xx` — Błąd spowodowany przez klienta. Np. `400 Bad Request` — błędnie sformułowane żądanie (np. błędy składniowe, brak pola `Host`), `401`

Unauthorized — brak dostępu do zasobu, czy 404 Not Found — serwer nie znalazł żądanego zasobu.
— 5xx — Błędy po stronie serwera, np. 500 Internal Server Error — błąd wewnętrzny serwera, czy 501 Not Implemented dana funkcjonalność w ogóle nie działa w systemie (np. metoda POST nie jest obsługiwana).

9.2.2.9. Metoda POST

Z technicznego punktu widzenia różnica między metodą GET a POST jest niewielka. Zarówno za pomocą jednej jak i drugiej możemy przesłać w zapytaniu dane na serwer oraz odebrać odpowiedź. W metodzie GET jedynym sposobem wysłania danych jest umieszczenie ich w URL-u (w części oznaczającej zapytanie). W metodzie POST dane wysyłane są jako część zasadnicza żądania (żądanie GET nigdy nie ma części zasadniczej). Te różnice są sprawą drugorzędną i wynikają z różnego zastosowania tych metod.

Metoda GET służy tylko do pobierania danych. Dane przesłane na serwer (w URL-u) powinny służyć tylko do tego, aby określić jakie dane chcemy od serwera otrzymać. Przykładem może być słownik online — musimy przesłać termin, którego tłumaczenie chcemy uzyskać.

Założeniem metody POST jest wykonanie pewnej akcji na serwerze. Może to być wysłanie wiadomości na forum, zarezerwowanie biletów, czy wysłanie wiadomości pocztowej.

Przykładem danych wysyłanych na serwer są dane z formularzy HTML.

Np. dane z poniższego formularza po wpisaniu przez użytkownika loginu kasia i kliknięciu przycisku spowodują wysłanie żądania GET z pierwszym wierszem następującym: GET /skrypt.php?login=kasia HTTP/1.1.

```
<form method="GET" action="skrypt.php" >
  Podaj login: <input name="login" type="text"/>
  <input type="submit"/>
</form>
```

Inaczej będzie w przypadku analogicznego formularza, ale z wykorzystaniem metody POST.

```
<form method="POST" action="skrypt.php" >
  Podaj login: <input name="login" type="text"/>
  <input type="submit"/>
</form>
```

W tym wypadku zostanie wysłane żądanie przedstawione poniżej.

```
POST /skrypt.php HTTP/1.1
Host: example.net
```

```
Content-Type: application/x-www-form-urlencoded
Content-Length: 11

login=kasia
```

Możliwość zapisania danych w URL-u powoduje m.in. że możemy utworzyć odnośnik do zasobu zawierający te dane. Każde użycie tego odnośnika skutkuje wysłaniem tych danych. O ile utworzenie odnośnika do znaczenia danego terminu w słowniku jest rozsądne, to odnośnik wysyłający po raz kolejny tę samą wiadomość na to samo forum byłby co najmniej irytujący. Poprawne wybranie metody w formularzu HTML pozwoli przeglądarce ostrzec nas, gdy próbujemy te same dane wysłać ponownie za pomocą metody POST. W przypadku poprawnie użytej metody GET nie ma powodu do takiego ostrzeżenia.

9.2.2.10. Metoda HEAD

Kolejną metodą HTTP jest HEAD, która używana jest zwykle do testowania. Odpowiedź na żądanie HEAD powinna różnić się od odpowiedzi na analogiczne żądanie GET tylko brakiem części zasadniczej. Jest to jedyna oprócz GET obowiązkowa metoda — pozostałe mogą nie być zaimplementowane[24].

9.2.2.11. Różnice między HTTP/1.1, a HTTP/1.0

.

Najważniejsze różnice między tymi wersjami protokołów omówiliśmy już w podrozdziałach 9.2.2.1 i 9.2.2.2. Dokładne omówienie tego tematu można znaleźć w pracy [31] oraz w sekcji 19.6.1 dokumentu [16].

Jeszcze jedną nieomówioną różnicą jest konieczność akceptowania przez serwery kompletnych URL-i w żądaniach, np. `GET http://server.example.org/plik.html HTTP/1.1` mimo, że żaden klient nie powinien generować takich żądań (jest to tłumaczone zgodnością z przyszłymi wersjami HTTP). Obecnie tego typu żądania kierowane są tylko do *Pośrednika HTTP* (ang. *HTTP proxy*).

Skondensowany opis protokołu HTTP z opisem wszystkich metod nagłówków, kodów odpowiedzi itp. znaleźć można w [61].

[24] Standard definiuje jeszcze metody OPTIONS, PUT, DELETE, TRACE oraz CONNECT, ale są one dużo rzadziej używane (i implementowane), więc nie będziemy się nimi tutaj zajmować.

9.3. Pytania i zadania

1. Napisz program umożliwiający użytkownikowi wysłanie maila za pomocą protokołu SMTP. Program dostaje jeden parametr — adres serwera SMTP, którego należy użyć i ewentualnie po dwukropku numer portu (domyślnie 25). Program pobiera od użytkownika następujące informacje: nadawcę, odbiorcę, temat i treść wiadomości. Jeżeli wysłanie się nie powiodło, to na standardowym wyjściu wyświetlana jest informacja o ostatnim żądaniu SMTP wysłanym do serwera i odpowiedź na nie. W przypadku użycia opcji `-v` program wyświetla informacje o wszystkich wysłanych żądaniach i odpowiedziach serwera.
2. Zmodyfikuj program z zadania poprzedniego tak, aby próbował przed wysłaniem maila zalogować się na serwerze metodą PLAIN lub LOGIN za pomocą loginu i hasła pobranych od użytkownika.
3. Zmodyfikuj programy z poprzednich zadań tak, aby umożliwiały przesyłanie załączników.
4. Napisz klienta protokołu POP3 pobierającego od użytkownika login oraz hasło i wyświetlającego liczbę listów oraz nadawców i tematy wszystkich wiadomości znajdujących się w skrzynce. Adres serwera (domenowy lub IP) zadany jest pierwszym parametrem wiersza poleceń. Opcjonalny drugi parametr oznacza numer portu (domyślnie używany jest port 110).
5. Zmodyfikuj program z poprzedniego zadania tak, żeby użytkownik po obejrzeniu tematów i nadawców wiadomości miał możliwość wyświetlenia treści wybranej wiadomości.
6. Napisz programy analogiczne do programów z dwóch poprzednich zadań, ale korzystające korzystające z protokołu IMAP.
7. Napisz klienta protokołu IMAP pobierającego od użytkownika login oraz hasło i wyświetlającego tematy i nadawców wiadomości zawierających w treści słowo zadane w argumencie wiersza poleceń.
8. Napisz program używający protokołu HTTP/1.1 dostający w parametrze wiersza poleceń odpowiedni URL. Program wysyła serwerowi żądanie HEAD dotyczące tego URL-a i wyświetla na standardowym wyjściu kod i komunikat z pierwszego wiersza odpowiedzi serwera (np. `200 OK`, `404 Not Found`).
9. Zmodyfikuj program z zadania poprzedniego tak, żeby wyświetlał również informacje o czasie ostatniej modyfikacji zasobu i o rodzaju oprogramowania serwera (o ile informacje te znajdują się w odpowiedzi serwera).
10. Napisz klienta protokołu HTTP/1.1 służącego do pobrania jednego pliku (URL zadany w wierszu poleceń). Jeżeli kod odpowiedzi serwera jest inny niż 200, to po program ma po prostu wyświetlić ten kod i towarzyszący mu komunikat.
11. Do programu z poprzedniego zadania dodaj obsługę przekierowań.

12. Zmodyfikuj program z dwóch poprzednich zadań tak, aby nie pobierał pliku jeżeli w jego katalogu roboczym istnieje plik o nazwie zadanej w URL-u nowszy niż na serwerze (użyć w nagłówku żądania pola `If-Modified-Since`).
13. Napisz prosty iteracyjny serwer HTTP udostępniający pliki ze swojego katalogu roboczego. Serwer wysyła pliki o rozszerzeniach texttttxt, `htm`, `html`, `pdf`, `ps`, `gif`, `jpeg` z odpowiednim typem MIME wysłanym w polu nagłówkowym `Content-Type` (typ MIME dobrać tylko na podstawie rozszerzenia, a nie sprawdzania zawartości pliku). Wszystkie pozostałe wysyłane są jako typ `application/octet-stream`. Po wysłaniu pliku serwer zamyka połączenie (zaznacza to też odpowiednim polem w nagłówku).
14. Zmodyfikuj program z zadania poprzedniego tak, aby w przypadku użycia przez klienta protokołu HTTP/1.1 serwer oczekiwał na dalsze żądania za pomocą tego samego połączenia. W takim przypadku serwer wysyła w nagłówku także rozmiar pliku.
15. * Napisz prosty serwer HTTP. Serwer udostępnia tylko pliki znajdujące się bezpośrednio w katalogu podanym pierwszym argumentem wiersza poleceń i mające rozszerzenie `html`, `css` lub `png`. Pliki wysyłane są z odpowiednio ustawionym w nagłówku odpowiedzi polem `Content-Type`. Pliki dostępne są za pomocą URL-a `http://`*`serwer`*`[:`*`port`*`]/pliki/`*`plik`*.

    Ponadto w przypadku użycia przez klienta URL-a `http://`*`serwer`*`[:`*`port`*`]/ciekawa-strona/` serwer przekierowuje żądanie na stronę, której URL jest argumentem opcji `-s` uruchomienia serwera. Przekierowanie odbywa się za pomocą odpowiedzi `302 Found`. Jeżeli serwer został uruchomiony bez opcji `-s` w wierszu poleceń, to zamiast przekierowania zostanie zwrócona odpowiedź `404 Not Found`.

    Dodatkowo pod adresem `http://`*`serwer`*`[:`*`port`*`]/admin/` dostępna jest strona z formularzem HTML umożliwiającym zmianę strony, na którą przekierowuje adres `/ciekawa-strona/`. Formularz zawiera dwa pola — pole tekstowe do wpisania adresu nowej strony i pole do wpisania hasła. Strona zmieniana jest tylko jeżeli podane zostanie poprawne hasło. Poprawne hasło pobierane jest od użytkownika przez serwer zaraz po jego uruchomieniu. Dane z formularza wysyłane są do serwera za pomocą metody POST. URL użyty w atrybucie `action` znacznika `form` powinien wskazywać także na stronę `/admin/`. Po odebraniu danych z formularza serwer powinien odesłać stronę HTML z informacją o tym, czy udało się zmienić „ciekawą stronę” na serwerze (zależy to tylko od poprawności hasła, nie jest sprawdzana poprawność wpisanego adresu — najwyżej przekierowanie przestanie poprawnie działać).

    To, czy wysłane żądanie z podanym zasobem `/admin/` powoduje wysłanie

strony z formularzem, czy powoduje próbę podmiany „ciekawej strony" i odesłania strony z informacją o tym czy się udało, zależy od użytej w żądaniu metody (GET lub POST).

Rozdział 10

Projektowanie i implementacja protokołów warstwy aplikacji

10.1. Podział danych na części **138**
    10.1.1. Przetwarzanie potokowe 139
10.2. Reprezentacja danych **139**
10.3. Kody odpowiedzi i informacje diagnostyczne **140**
10.4. Stany protokołów **140**
    10.4.1. Rozszerzanie protokołu 141
10.5. Wykorzystanie istniejących rozwiązań **142**
10.6. Różne uwagi **143**
10.7. Pytania i zadania **143**

Podczas pisania programów sieciowych musimy zawsze rozwiązać typowy problem. Jeżeli wiemy już jakie dokładnie zadanie ma być rozwiązane za pomocą naszego programu, to musimy przemyśleć, jak to zadanie ma być rozwiązane. W przypadku programów sieciowych jedną z najważniejszych kwestii jest teraz określenie protokołu przesyłania danych — ich rozmiaru, formatu, kolejności.

Rozpatrzmy następujący prosty przykład. Chcemy napisać program, który po podaniu przez użytkownika nazwy pliku pobiera ten plik ze zdalnego serwera. Pominiemy w rozważaniach kwestie dotyczące sposobu interakcji programów z użytkownikiem czy z systemem operacyjnym. Skupimy się na przesyłanych danych. Musimy przesłać serwerowi nazwę pliku, a następnie odebrać plik.

10.1. Podział danych na części

Jednym z problemów, którym musimy się zająć, jest zaznaczenie końca pewnej porcji danych. Protokół UDP sam się tym zajmuje, ale większość protokołów sieciowych opartych jest na połączeniowej usłudze strumieniowej protokołu TCP. Strumieniowość oznacza tutaj, że dane traktowane są jako ciąg bajtów i sami musimy narzucić im jakąś strukturę. W naszym przypadku podstawowym problemem jest to, jak serwer ma rozpoznać, że przeczytał już całą nazwę pliku. Możemy to rozwiązać na różne sposoby. Możemy przesłać przed nazwą pliku długość tej nazwy, możemy też zaznaczyć koniec danych jakimś znakiem bądź sekwencją znaków[1], które nie mogą wystąpić w samych danych, wreszcie możemy też po prostu zaznaczyć koniec przesyłanych danych na poziomie TCP[2]. Oczywiście identyczny problem dotyczy przesyłania danych w drugą stronę.

W protokole HTTP są używane różne sposoby określania końca danych. Jednym jest nagłówek `Content-Length` — domyślnie w HTTP/1.1. Innym po prostu zamknięcie połączenia przez serwer po odesłaniu odpowiedzi — domyślnie w protokole HTTP/1.0.

Innym przykładem są protokoły POP3 i SMTP i koniec wprowadzanej wiadomości zaznaczony wierszem zawierającym samą kropkę.

Czasem możemy też po prostu przyjąć, że dane mają określoną liczbę bajtów. Zauważmy, że np. wszystkie oryginalne komendy protokołów pocztowych POP3 i SMTP są czteroznakowe. Można przypuszczać, że było to związane właśnie z prostotą implementacji odbierania tych komend.

[1] Np. typowym dla protokołów tekstowych końcem wiersza — czyli sekwencją CRLF.

[2] Nie możemy oczywiście tego zrobić za pomocą funkcji `close`, bo nie moglibyśmy potem odebrać odpowiedzi. Trzeba to zrobić za pomocą funkcji `shutdown` z argumentem `SHUT_WR`.

10.1.1. Przetwarzanie potokowe

Jeżeli protokół umożliwia wysłanie za pomocą pojedynczego połączenia wielu par żądanie-odpowiedź, oraz pary te są od siebie niezależne, to w implementacji można zastosować technikę przetwarzania potokowego (ang. *pipelining*). Polega ona na przesyłaniu przez klienta wszystkich żądań na raz, bez czekania na odpowiedzi serwera.

W naszym przypadku możemy zmienić protokół tak, żeby wysyłać w jednym żądaniu wiele nazw plików i oczekiwać wielu plików w odpowiedzi. Serwer może wtedy przetwarzać kolejne zapytanie od razu po poprzednim — nie musi czekać aż klient odbierze i przetworzy odpowiedź oraz wyśle kolejne żądanie. Poza tym możliwe jest, że ciąg krótkich żądań zostanie przesłany nawet jako pojedynczy segment TCP. Wszystko to może znacznie przyspieszyć działanie programów. Wiąże się to też z pewnymi problemami implementacyjnymi. Jeżeli serwer przetworzy mniej żądań niż wysłał klient i zamknie połączenie, to część żądań może przyjść już po zamknięciu połączenia. Odpowiedzią na nie będzie segment RST powodujący zgłoszenie błędu po stronie klienta. Jeżeli segment ten dotrze do warstwy TCP klienta, zanim klient odbierze z tej warstwy wszystkie odpowiedzi, to podczas odbierania kolejnej zostanie zgłoszony błąd i klient nie powinien już próbować czytać dalszych danych. W rezultacie klient może odebrać nawet mniej odpowiedzi niż poprawnie wygenerował serwer. Poprawna implementacja powinna wymagać od serwera przeczytania wszystkich danych klienta, nawet jeżeli ich nie przetworzy, a od klienta czytania danych od serwera możliwie szybko — najlepiej żeby klient oczekiwał na dane równocześnie z ich wysyłaniem (funkcja `select` lub wątki) w celu możliwie szybkiego wykrycia zamknięcia połączenia przez serwer. Po wykryciu zamknięcia połączenia klient nie wysyła już dalszych żądań.

Przetwarzanie potokowe może być zastosowane np. w protokole HTTP.

10.2. Reprezentacja danych

Kolejną sprawą może być różny sposób reprezentacji tych samych danych w różnych systemach wymieniających je. Załóżmy na przykład, że w jednym z systemów pliki tekstowe kodowane są za pomocą standardu ISO 8859-2, a w drugim CP-1250 (Windows-1250). Jeżeli chcielibyśmy, żeby użytkownicy naszego programu bez problemu wymieniali pliki pomiędzy tymi systemami bez zajmowania się konwertowaniem ich za pomocą jakiegoś osobnego programu, to musimy jakoś to przewidzieć w naszym protokole. Możemy np. ustalić, że wszystkie pliki przesyłane są w pewnym ustalonym kodowaniu, np. UTF-8. Wtedy strona wysyłająca będzie musiała przekonwertować plik ze swojego kodowania na UTF-8, a druga — z UTF-8 na swój system kodo-

wania. Możemy też zawsze razem z plikiem przesyłać dodatkową informację o tym, jakie jest użyte kodowanie. Możemy też połączyć oba rozwiązania i mieć możliwość przesyłania informacji o kodowaniu, a w razie jej braku przyjąć jakieś kodowanie domyślne.

To ostatnie rozwiązanie jest np. przyjęte w protokole HTTP. W HTTP dodatkowo występuje możliwość negocjowania kodowania, w którym będą przesyłane dane tekstowe.

Dokładny sposób, w jaki protokoły powinny postępować z kodowaniem przesyłanych danych tekstowych, opisany jest w dokumencie [1].

Analogiczną sytuację w przypadku systemów binarnych mamy choćby z kolejnością bajtów w przesyłanych danych. Tutaj jednak rozwiązaniem raczej nie jest przesyłanie informacji o kolejności. Raczej wszystkie programy sieciowe korzystają po prostu z sieciowej kolejności bajtów (patrz podrozdział 3.1.1).

10.3. Kody odpowiedzi i informacje diagnostyczne

Nasz protokół musi oczywiście uwzględniać sytuacje, gdy nie jest w stanie z jakiegoś powodu spełnić żądania klienta. Protokoły transportowe zwracają zawsze w odpowiedzi informację o typie tej odpowiedzi — czy są to np. żądane dane, czy odesłanie klienta w inne miejsce, czy informacja o błędzie.

W przypadku protokołów tekstowych, opartych na TCP, taka informacja składa się zwykle z dwóch części — pierwsza jest jakimś krótkim kodem, np. liczbą, znakiem `+`, czy krótkim napisem, druga — informacją tekstową w ogóle nie używaną przez programy użytkowe (co najwyżej przekazywaną użytkownikowi). Pierwsza jest po to, żeby klient mógł poprawnie na nią zareagować, druga — w celach diagnostycznych. Można ją wykorzystać np. podczas sprawdzania działania protokołu za pomocą programu `telnet`.

Kody odpowiedzi często podzielone są na grupy w zależności od ich znaczenia.

Warto przyjrzeć się kodom odpowiedzi generowanym przez serwery różnych znanych protokołów, choćby protokołów pocztowych czy HTTP.

10.4. Stany protokołów

W celu łatwiejszego opisu protokołu używa się często pojęcia stanu protokołu. Przykłady takich stanów były podane w podrozdziale 9.1.3.2. Innym przykładem jest protokół TCP. Nie jest to protokół warstwy aplikacji, ale zasada jest ta sama. To, jaka akcja jest możliwa do wykonania, zależy od stanu, w jakim znajduje się protokół.

10.4.1. Rozszerzanie protokołu

Warto już na etapie projektowania protokołu zastanowić się, czy łatwo będzie dodać do niego rozszerzenia, jeżeli okaże się, że będą przydatne.

W przypadku danych tekstowych często rozszerzenie protokołu można uzyskać po prostu przez dopisanie pewnych danych do standardowej wiadomości, np. dodanie nowego pola nagłówkowego. Jednak czasem nie będziemy mogli tak zrobić, jeżeli specyfikacja naszego protokołu będzie wymagała przekazania informacji o błędzie w przypadku nieznanego nagłówka, mimo że zignorowanie go byłoby akceptowalnym rozwiązaniem. Jeżeli teraz chcielibyśmy dodać takie pole usprawniające pracę nowszej wersji protokołu, to wymagałoby to wyrzucenia wszystkich istniejących implementacji i zastąpienie ich nowymi. Jeżeli nasz protokół zyskał jakąkolwiek popularność — naszego programu używa choćby kilkoro naszych znajomych — to będzie to niezwykle trudne.

Innym sposobem jest nadanie nowego znaczenia przesyłanym danym. Tak było np. z protokołami pocztowymi i wprowadzeniem standardu MIME[3]. Pierwsza wersja poczty pozwalała na przesyłanie jedynie plików tekstowych. MIME pozwala na dużo większą swobodę (patrz podrozdział 9.1.2.2). Dane w formacie MIME są zakodowane w postaci zwykłego tekstu. Powszechnie używany i działający jeszcze przed wprowadzeniem MIME protokół POP3 w ogóle nie rozumie i nie musi rozumieć tego formatu — dane są odbierane w identyczny sposób jak wcześniej, a interpretacją ich zajmuje się program pocztowy. Protokół IMAP z kolei jest nowszy i MIME pozwala mu na lepsze wykorzystanie zasobów.

Dane w formacie binarnym zazwyczaj mają bardziej rygorystycznie narzuconą strukturę niż dane tekstowe. Jeżeli np. przyjmiemy w naszym protokole opartym o protokół UDP, że przesyłany datagram ma z góry określony rozmiar lub przed danymi jest nagłówek o określonym rozmiarze, to dodanie dodatkowej funkcjonalności do naszego protokołu i współdziałanie z istniejącymi implementacjami może być trudne.

Zobaczmy jak sprawę tę rozwiązały protokoły takie jak IPv4, IPv6 i TCP. W protokole IPv4 mamy dodatkowe pole *opcje*, którego długość określona jest polem IHL i jest ograniczona do 40 bajtów. Podobne rozwiązanie zastosowane jest w nagłówku TCP. Elastyczniejsze rozwiązanie jest zastosowane w IPv6 — nagłówek zawiera pole *następny nagłówek*, który informuje, że po nagłówku pojawią się albo dane protokołu wyższej warstwy albo *nagłówek dodatkowy* protokołu IPv6. Nagłówek dodatkowy też ma pole *następny nagłówek* tak samo interpretowany. Umożliwia to dodanie dowolnej liczby nagłówków dodatkowych. Co prawda wymienione protokoły nie są protoko-

[3] Tak naprawdę, to jest to połączenie obu sposobów — przekazywanie wiadomości w formacie MIME wymaga także dodatkowych nagłówków.

łami warstwy aplikacji, ale w przypadku użycia protokołu UDP w warstwie transportowej można zastosować podobne rozwiązania.

10.5. Wykorzystanie istniejących rozwiązań

Zaprojektowanie protokołu działającego poprawnie, przy założeniu że nie wystąpią jakieś wyjątkowo nietypowe sytuacje, nie jest trudne. W rzeczywistości jednak przewidzenie wszystkich możliwych sytuacji, z którymi nasz protokół będzie musiał sobie radzić, jest trudne. Im bardziej protokół jest rozbudowany, ma więcej poleceń, opcji, kodów odpowiedzi, tym jest to trudniejsze. Narzędziami, które mogą nam trochę pomóc są opisane już stany (podrozdział 10.4) oraz wykorzystanie automatów skończonych do opisania wszelkich możliwych zachowań naszego protokołu w różnych sytuacjach [27].

Jednak tak naprawdę pierwsza porada dotycząca tworzenia nowego protokołu powinna być taka, żeby tego po prostu unikać.

Po pierwsze należy rozważyć wykorzystanie istniejącego protokołu. Istniejące protokoły są zazwyczaj świetnie przetestowane, a ewentualne problemy z nimi są przynajmniej dobrze znane. Warto zastanowić się, czy żaden z istniejących już protokołów nie rozwiązuje naszych problemów. Kolejną zaletą użycia sprawdzonego protokołu jest duża szansa na istnienie bibliotek[4] czy innych narzędzi wspomagających tworzenie aplikacji dla tego protokołu.

Jeżeli żaden z protokołów nie jest idealnie dopasowany do naszych potrzeb, to można rozważyć rozszerzenie istniejącego protokołu o nowe opcje.

Można też, jeżeli nasz program nie potrzebuje wszystkich opcji używanych przez oryginalny protokół, rozważyć uproszczenie go. Należy też wtedy uważać z używaniem odpowiedniej terminologii w odniesieni do naszego programu. Nie powinniśmy mówić, że nasz program obsługuje dany protokół, jeżeli nie spełnia on pewnych minimalnych wymagań narzuconych przez specyfikację protokołu i zachowuje się nieprawidłowo w pewnych sytuacjach.

Nawet jeżeli nie będziemy w żaden sposób używać istniejącego protokołu, to dobrze jest rozważyć użycie jakiegoś standardowego formatu wiadomości przesyłanej przez nasz protokół (np. rfc822 bazującego na [12]; nazwy nagłówków używane przez różne protokoły znajdują się w [42]), czy formatu przesyłanych danych (np. XML — ang. *eXtensible Markup Language*). Zaletami znowu jest istnienie bibliotek i innych narzędzi.

Warto także stosować się do różnych zaleceń dotyczących formatów danych takich jak np. data i czas[5].

---

[4] Warto tu wspomnieć choćby o bibliotece libcurl związanej z projektem cURL [63] implementującej wiele protokołów warstwy aplikacji w tym wszystkie opisane w tej książce.

[5] Te zalecenia znajdują się w [30].

W każdym razie oparcie się o istniejący protokół jest na pewno dobrym pomysłem — i pierwszym, który powinniśmy wziąć pod uwagę.

10.6. Różne uwagi

— Dobrze jest, jeżeli w protokole nie ma zbyt dużej asymetrii, tzn. klient nie powinien mieć możliwości spowodowania niewielkim wysiłkiem wykonania obciążającej serwer akcji. Taka niesymetryczność ułatwia ataki DoS (ang. *Denial of Service* — odmowa usługi).
— Warto wziąć pod uwagę wpływ zastosowania takich mechanizmów jak NAT czy firewall na działanie naszego protokołu. Problemem może być zapisywanie przez program do swoich danych adresów IP czy numerów portów. Konwerter NAT zmieni adresy i porty w pakietach IP, a w danych nie, jeżeli nie będzie miał wiedzy o naszym protokole. Problemem jest też próba otwierania przez klienta konkretnych portów i oczekiwanie na połączenia na nich[6].
— Warto też stosować się do takiej zasady, żeby nasze programy (zarówno klient jak i serwer) wysyłały wszystkie dane całkowicie zgodne ze specyfikacją, natomiast w odbieranych danych pozwalały na pewne drobne odstępstwa od specyfikacji. Przykładem takiego zachowania może być akceptacja wierszy kończących się znakiem LF zamiast sekwencji CRLF, czy akceptowanie dowolnego ciągu spacji i tabulatorów tam, gdzie specyfikacja wymaga pojedynczej spacji.

10.7. Pytania i zadania

1. Napisz dwa programy — klienta i serwer TCP do wykonywania zdalnego dodawania.
   — Klient dostaje w kolejnych argumentach wiersza poleceń dwie liczby całkowite z zakresu $0 \ldots 2^{31} - 1$ (i liczby i ich suma są możliwe do zapisania na czterech bajtach), adres i port serwera. Klient wysyła osiem bajtów danych — liczby zapisane na czterech bajtach w sieciowej kolejności bajtów, a następnie odbiera cztery bajty wyniku — sumę tych liczb. Suma wyświetlana jest na standardowym wyjściu.
   — Serwer iteracyjny dostaje w argumencie wiersza poleceń numer portu, na którym ma nasłuchiwać. Po zaakceptowaniu połączenia serwer odbiera osiem bajtów danych (dwie czterobajtowe liczby), odsyła cztery bajty wyniku (suma tych liczb) i kończy połączenie.

[6] Taka sytuacja ma np. miejsce w FTP.

2. Zmodyfikuj programy z zadania poprzedniego tak, żeby liczby były przesyłane w formacie tekstowym — klient wysyła kolejno cyfry pierwszej liczby, pojedynczą spację i cyfry drugiej liczby. Serwer odsyła cyfry wyniku i kończy połączenie.
3. Zmodyfikuj programy z zadania poprzedniego tak, żeby za pomocą jednego połączenia klient mógł wysłać wiele kompletów danych. Każdy komplet danych i każdy wynik przesyłany jest w osobnym wierszu zakończonym sekwencją znaków CRLF (`'\r\n'`).
4. Niech serwer w jednym połączeniu odpowiada tylko na pięć zapytań. Jaki problem może się pojawić, jeżeli klient od razu wyśle więcej niż pięć zapytań, następnie spróbuje odebrać odpowiedzi serwera, a serwer od razu po odesłaniu pięciu odpowiedzi zamknie połączenie? Jeżeli program nie jest odporny na ten problem, to zmodyfikuj odpowiednio programy klienta i serwera.
5. Zaprojektuj i zaimplementuj protokół do zdalnej edycji plików tekstowych. Protokół ma umożliwiać:
   — odczytanie liczby wierszy pliku,
   — odczytanie zadanego bloku wierszy,
   — zastąpienie zadanego bloku wierszy innym zadanym.

   Załóż, że pliki zawierają tylko kody ASCII — serwer ma uniemożliwiać edytowanie innych.
6. Spróbuj zmienić protokół z poprzedniego zadania tak, żeby dodać możliwość obsługi polskich znaków.
7. Dodaj do protokołu i zaimplementuj mechanizm analogiczny do `If-Modified-Since` i `If-Unmodified-Since` z protokołu HTTP.

Dodatek A

Jak to jest w Pythonie?

A.1. Interfejs gniazd **146**
A.1.1. Wyjątki 146
A.1.2. Klasa `socket.socket` 147
A.1.2.1. Metody inicjujące/finalizujące 147
A.1.2.2. Metody wysyłające/odbierające . . . 149
A.1.2.3. Ustawienia i opcje gniazd 150
A.1.3. Inne funkcje modułu `socket` 151
A.2. Serwery i demony **153**
A.3. Moduły wyższego poziomu **157**
A.3.1. Sieciowe elementy modułu `multiprocessing` . 158
A.3.2. Obiekty rozproszone w module `multiprocessing` 159
A.3.3. Przegląd innych modułów sieciowych 163

W niniejszym dodatku zakładamy, że Czytelnik zna już język Python i choć trochę potrafi się w nim poruszać — jeśli nie, to możemy polecić wiele materiałów tak papierowych jak i sieciowych [14, 34, 36, 41, 43, 64, 65, 9]. Zakładamy w szczególności, że interfejs plikowy Pythona jest Czytelnikowi znany, będziemy bowiem się do niego odwoływać.

Używać będziemy tutaj wersji 2.6.6 Pythona, ale wszystkie przykłady powinny działać także z innymi wersjami Pythona 2 — a po drobnych zmianach kosmetycznych także z Pythonem 3.

Python jest wieloparadygmatowym językiem wysokiego poziomu. Jest językiem właściwie całkowicie obiektowym (obiektami są wszelkie dane, ale także funkcje czy moduły) jednocześnie pozwalając programować funkcyjnie oraz strukturalnie/proceduralnie. Dołączona do Pythona obszerna biblioteka standardowa pozwala na oprogramowywanie wielu różnych zagadnień łatwo i bez pomocy dodatkowych bibliotek. Co więcej, tak powstałe programy są wysoce przenośne pomiędzy różnymi maszynami i systemami operacyjnymi.

Nie inaczej jest z interfejsem gniazd i programowaniem sieciowym. Ponadto — jeśli chodzi o przenośność — pomimo Uniksowego interfejsu gniazd w Pythonie, wszystkie jego elementy działają także pod MS Windows (oraz pod wieloma innymi systemami).

A.1. Interfejs gniazd

Za interfejs gniazd w Pythonie odpowiada standardowy moduł `socket`. Wiele funkcji tego modułu odpowiada Uniksowym funkcjom systemowym obsługującym gniazda (patrz rozdziały 3–7), jednakże są one także zaimplementowane (w większości) dla innych systemów operacyjnych (jak MS Windows). Ponadto, wykorzystują właściwości języka i jego standardowe wysokopoziomowe struktury danych, by ułatwić korzystanie z gniazd i nie obarczać programisty myśleniem niskopoziomowym.

A.1.1. Wyjątki

W Pythonie ważnym instrumentem kontroli poprawności działania programu są wyjątki[1]. Moduł `socket` korzysta oczywiście z wyjątków standardowych Pythona, lecz także definiuje cztery własne[2]:

---

[1] Większość nieprawidłowości w działaniu (ale też sytuacje prawidłowe, lecz właśnie wyjątkowe) jest w Pythonie obsługiwana przez wyjątki, a nie przez specjalne wartości funkcji (jak to ma miejsce w przypadku funkcji systemowych).

[2] Zakładamy w niniejszym dodatku, że wszystkie moduły importujemy przez

```
import nazwa
```

więc odwołujemy się do komponentów modułu w sposób w pełni kwalifikowany: *`nazwa.funkcja`*, *`nazwa.klasa`*, *`nazwa.STAŁA`* itp.

— `socket.error`, który jest zgłaszany, gdy wystąpią ogólne problemy związane z gniazdami, które nie pasują do standardowych wyjątków ani do żadnego z poniższych;
— `socket.gaierror`, który jest zgłaszany, gdy wystąpią błędy w funkcjach `socket.getaddrinfo` oraz `socket.getnameinfo`;
— `socket.herror`, który jest zgłaszany w pozostałych funkcjach związanych z adresami (w C odpowiada to funkcjom używającym `h_errno` do zgłoszenia błędu — p. str. 47);
— `socket.timeout` używany do sygnalizacji upływu limitu czasu (timeoutu).

A.1.2. Klasa `socket.socket`

Kluczowym elementem Pythonowego interfejsu gniazd jest klasa `socket.socket`, której konstruktor ma następującą postać:

```
socket.socket(family=socket.AF_INET,
              type=socket.SOCK_STREAM,
              proto=0)
    # zwraca obiekt typu socket.socket
```

Jak widać powyżej, każdy z trzech argumentów może być opuszczony, a odpowiadają one dokładnie argumentom funkcji systemowej `socket` (p. str. 56)[3]. Podstawową różnicą względem wspomnianej funkcji jest zwracana wartość — tutaj nie jest zwracany deskryptor gniazda, lecz oczywiście obiekt konstruowanej klasy.

Oprócz podanych wyżej domyślnych wartości parametrów można używać także innych stałych z modułu `socket` o nazwach identycznych z nazwami stałych bibliotecznych języka C (p. str. 56).

A.1.2.1. Metody inicjujące/finalizujące

Opisane tutaj metody klasy `socket.socket` służą do różnego rodzaju obsługi nawiązywania i rozwiązywania połączeń miedzy gniazdami. Odpowiadają one dokładnie Uniksowym funkcjom systemowym o tych samych nazwach (patrz rozdziały 4–6), wprowadzają jednak wiele udogodnień zgodnych z duchem Pythona. Są to poniższe funkcje (przy założeniu, że `s` jest obiektem klasy `socket.socket`).

```
s.connect(address)
    # zwraca None
s.close()
```

[3] Całkiem analogicznie jest z innymi funkcjami modułu `socket`, aczkolwiek Python ich interfejsy w wielu miejscach rozszerza lub uprzyjaźnia.

```
        # zwraca None
    s.shutdown(how)
        # zwraca None
    s.bind(address)
        # zwraca None
    s.listen(backlog)
        # zwraca None
    s.accept()
        # zwraca nowy obiekt typu socket.socket
```

W powyższych funkcjach `backlog` jest długością kolejki nieobsłużonych połączeń (p. str. 79), zaś `how` określa sposób zamknięcia (najlepiej stałymi `socket.SHUT_RD`, `socket.SHUT_WR`, `socket.SHUT_RDWR` — p. str. 83).

Natomiast argument `address` wymaga nieco więcej wyjaśnień. Podobnie, jak jest to w Uniksowym interfejsie funkcji systemowych języka C, podawany jest tutaj adres w formacie zależnym od rodziny (domeny) adresów. Jednakże, adres ów konstruowany jest dużo prościej, i w przypadku rodziny `socket.AF_INET` (która w niniejszym skrypcie nas interesuje przede wszystkim) jest to po prostu Pythonowa para (dwuelementowa krotka), której elementy są opisane poniżej.

— Pierwszy komponent to adres komputera, który może być napisem — będącym albo jego adresem IP (w tradycyjnej formie czterech dziesiętnych liczb całkowitych oddzielonych kropkami, na przykład: `'212.182.0.171'`) albo jego adresem domenowym (na przykład: `'matrix.umcs.lublin.pl'` — wtedy przekształcenie na adres IP i odpytywanie serwerów DNS wykonywane są automatycznie). Może też przyjąć jedną z dwóch specjalnych form: napis pusty odpowiadający stałej bibliotecznej `INADDR_ANY` języka C oraz napis `'<broadcast>'` odpowiadający stałej bibliotecznej `INADDR_BROADCAST` języka C.
— Drugi to numeru portu, który jest tutaj zwykłą Pythonową liczbą całkowitą.

Za wszelkie konwersje i poszukiwania odwzorowań odpowiada już sam Python.

Warto wspomnieć tutaj jeszcze o jednej funkcji pomocniczej łączącej funkcjonalność konstruktora z metodą `connect`:

```
    socket.create_connection(address[, timeout])
        # zwraca nowy obiekt typu socket.socket
```

Funkcja ta tworzy nowe gniazdo i od razu próbuje wykonać połączenie z podanym adresem. Drugi parametr może być opuszczony — przyjmowany jest wtedy domyślny limit czasu.

A.1.2.2. Metody wysyłające/odbierające

W przeciwieństwie do gniazd realizowanych na poziomie systemu operacyjnego Unix (reprezentowanych przez zwykłe deskryptory plikowe), Python odróżnia obiekty gniazdowe od zwykłych obiektów plikowych i nie udostępnia dla gniazd tradycyjnych metod odczytu i zapisu (typu `write` oraz `read`). Do realizowania komunikacji potrzebne są metody specjalne — najważniejsze z nich pokazane są poniżej (tu także zakładamy, że `s` jest obiektem klasy `socket.socket`).

```
s.recv(bufsize, flags=0)
    # zwraca string
s.recvfrom(bufsize, flags=0)
    # zwraca parę (string, address)
s.send(string, flags=0)
    # zwraca liczbę wysłanych bajtów
s.sendall(string, flags=0)
    # zwraca None
s.sendto(string, flags=0, address)
    # zwraca liczbę wysłanych bajtów
```

Parametry i wartości zwracane w podanych tu funkcjach:

— `bufsize` to liczba bajtów, które należy odebrać z gniazda;
— `flags` to znaczniki analogiczne do tych używanych na poziomie funkcji systemowych (p. str. 93);
— `string` to ciąg bajtów do wysłania lub odebrany — w postaci Pythonowego napisu (ciągu znaków);
— `address` to adres gniazda po przeciwnej stronie połączenia w formacie odpowiednim dla rodziny protokołów (dla rodziny `socket.AF_INET` jest to — jak poprzednio — para `(komputer, port)`.

Funkcje te działają analogicznie do odpowiednich systemowych, zwalniając jednak programistę z konieczności zajmowania się rezerwacją i zwalnianiem pamięci na bufor danych.

Kilka z omówionych do tej pory składników modułu `socket` przedstawiają listingi A.1–A.2 stanowiące parę klient-serwer nieco zmodyfikowanej usługi echo.

Listing A.1. Przykładowy prosty klient w Pythonie

```
  # coding: utf-8
2
  import socket
4 import time
```

```
6 ADRES = ('localhost', 1817)

8 for n in xrange(1, 100):
      s = socket.socket()
10    s.connect(ADRES)
      wyslano = s.send(str(n)*n)
12    print 'Wysłano␣bajtów:', wyslano
      dane = s.recv(256)
14    print 'Odebrano:', repr(dane)
      s.close()
16    time.sleep(1)
```

Listing A.2. Przykładowy prosty serwer w Pythonie

```
  import socket
2
  ADRES = ('', 1817)
4
  s = socket.socket()
6 s.bind(ADRES)
  while True:
8     s.listen(1)
      pol, adr = s.accept()
10    print 'Odbieram␣z', adr
      while True:
12        dane = pol.recv(256)
          if not dane:
14            break
          print 'Odebrano:␣', repr(dane)
16        pol.send(dane)
      pol.close()
```

A.1.2.3. Ustawienia i opcje gniazd

Także i tutaj mamy wiele metod klasy `socket.socket` odpowiadających nazwą i działaniem funkcjom systemowym, aczkolwiek są i takie, które odpowiedników nie mają (znowu: `s` jest obiektem klasy `socket.socket`).

```
s.getpeername()
    # zwraca address
s.getsockname()
    # zwraca address
s.getsockopt(level, optname, buflen)
    # zwraca liczbę całkowitą
    # lub zakodowaną strukturę języka C
```

```
s.setsockopt(level, optname, value)
    # zwraca None
s.settimeout(value)
    # zwraca None
s.gettimeout()
    # zwraca liczbę zmiennoprzecinkową lub None
s.setblocking(flag)
    # zwraca None
```

Dwie pierwsze metody (`getpeername` oraz `getsockname`) zwracają odpowiednio adresy[4] gniazda zdalnego i gniazda lokalnego z bieżącego połączenia.

Metody `getsockopt` oraz `setsockopt` odwzorowują dość dokładnie odpowiednie wywołania systemowe (rozdział 7) i w związku z tym ich użycie może być dość skomplikowanie. W obu używać można stałych z modułu `socket` nazwanych tak samo, jak odpowiednie stałe biblioteki języka C. Jeżeli wartością opcji jest liczba całkowita, wtedy należy opuścić `buflen` w `getsockopt`, a w `setsockopt` podać jako `value` ową liczbę. Jednakże, gdy wymagana jest dla danej opcji struktura języka C, wtedy `value` musi być zakodowaną jako napis strukturą języka C, natomiast `buflen` spodziewaną długością takiej struktury. Do obsługi takich danych służy standardowy moduł Pythona o nazwie `struct` (więcej w dokumentacji tego modułu [65]).

W końcu ostatnie trzy funkcje z podanych powyżej (`settimeout`, `gettimeout`, `setblocking`) odpowiadają za limit czasu oczekiwania na ukończenie operacji dla gniazda. Limit ten może być zerowy (operacja musi wykonać się natychmiast, inaczej zgłasza wyjątek), dodatni (wyrażony zmiennoprzecinkową liczbą sekund) lub nieskończony (symbolizowany w Pythonie przez `None` — gniazdo znajduje się wtedy w trybie blokującym i jest to domyślny tryb gniazda). Metody `settimeout` oraz `gettimeout` służą odpowiednio do ustawiania i sprawdzania limitu dla danego gniazda, natomiast `s.setblocking(True)` jest równoważne wywołaniu `s.settimeout(None)`, a `s.setblocking(False)` jest równoważne wywołaniu `s.settimeout(0)`.

A.1.3. Inne funkcje modułu `socket`

Omawiany moduł zawiera także inne funkcje odpowiadające funkcjom systemowym oraz kilka funkcji specyficznych dla owego modułu.

Poniższe funkcje odpowiadają za sprawdzenie i ustawienie domyślnego limitu czasu operacji dla nowo tworzonych gniazd. Normalnie, nowo tworzone gniazda są w trybie blokującym, co oznacza, że domyślną wartością tego limitu jest w Pythonie `None`.

[4] W formacie zależnym od rodziny adresów — dla rodziny `socket.AF_INET` opisanym w podrozdziale A.1.2.1.

```
socket.getdefaulttimeout()
    # zwraca domyślny limit czasowy dla nowo
    # tworzonych gniazd
socket.setdefaulttimeout(timeout)
    # zwraca None
```

Kolejne funkcje sygnalizujemy tylko, bo ich działanie jest albo oczywiste, albo takie jak odpowiednich funkcji systemowych czy też bibliotecznych języka C. Co więcej, wiele z nich jest potrzebnych bardzo rzadko, ze względu na to, że Python sam dokonuje znajdowania adresów oraz konwersji różnych parametrów (i tu także `s` jest obiektem klasy `socket`). Nie są to poza tym jeszcze wszystkie funkcje z modułu `socket`.

```
s.fileno()
    # zwraca plikowy deskryptor gniazda (do użycia
    # na przykład w funkcji select.select
    # (podrozdział poniżej)
socket.getaddrinfo(host, port,
                   family=0, socktype=0,
                   proto=0, flags=0)
    # zwraca listę krotek pięcioelementowych
    # opisujących adresy związane z połączeniem
socket.gethostname()
    # zwraca nazwę komputera wykonującego program
socket.getfqdn([name])
socket.gethostbyname(hostname)
socket.gethostbyname_ex(hostname)
socket.gethostbyaddr(ip_address)
    # funkcje szukające nazw i adresów komputerów
socket.getservbyname(servicename[, protocolname])
socket.getservbyport(port[, protocolname])
    # funkcje tłumaczące pomiędzy nazwą
    # i numerem portu/serwisu
socket.ntohl(x)
socket.ntohs(x)
socket.htonl(x)
socket.htons(x)
    # konwersje liczb pomiędzy sieciową a hostową
    # kolejnością bajtów
socket.inet_aton(ip_string)
socket.inet_ntoa(packed_ip)
socket.inet_pton(address_family, ip_string)
socket.inet_ntop(address_family, packed_ip)
    # konwersje adresów między różnymi formatami
```

A.2. Serwery i demony

Inne funkcje systemowe zwykle także są dostępne z poziomu Pythona. Niniejszy podrozdział przedstawia kilka takowych — jednakże, w odróżnieniu od wyżej omówionych są one przede wszystkim dostępne w systemach Uniksowych. Opisane tu funkcje ogólnosystemowe (nie dotyczące specyficznie gniazd) przydatne są przede wszystkim przy konstrukcji serwerów-demonów, w szczególności współbieżnych (całkiem analogicznie do tych z podrozdziału 8.1). Interesujące nas funkcje znajdują się w modułach `os`, `signal` oraz `select`.

Moduł `os` zawiera interfejs do podstawowych funkcji systemowych. Przydatne przy programowaniu sieciowym mogą być następujące plikowe operacje niskopoziomowe.

```
os.chdir(path)
    # zwraca None
os.umask(mask)
    # zwraca poprzednią maskę
os.tmpfile()
    # zwraca obiekt plikowy (tymczasowy) otwarty
    # do zapisu
```

Aby tworzyć serwery współbieżne wieloprocesowe oraz demony potrzebujemy jednak jeszcze zarządzania procesami. Do tego służyć mogą poniższe funkcje.

```
os.setsid()
    # zwraca None
os._exit(status)
    # nic nie zwraca, bo kończy proces
    # (niskopoziomowo)
os.execl(path, arg0, arg1, ...)
os.execle(path, arg0, arg1, ..., env)
os.execlp(file, arg0, arg1, ...)
os.execlpe(file, arg0, arg1, ..., env)
os.execv(path, args)
os.execve(path, args, env)
os.execvp(file, args)
os.execvpe(file, args, env)
    # niczego nie zwracają, bo wczytują
    # i uruchamiają nowy kod
os.fork()
    # zwraca 0 w potomku, a PID potomka w rodzicu
os.kill(pid, sig)
    # zwraca None
```

```
os.wait()
os.waitpid(pid, options)
os.wait3([options])
os.wait4(pid, options)
    # zwracają status i informacje
    # o zakończonym potomku
```

Nie wymagają owe funkcje wyjaśnień, bo naśladują one wiernie działanie i interfejs funkcji systemowych i bibliotecznych (z rozdziału 8). Jako opcje w wywołaniach funkcji czekających (dokładniej: ostatnich trzech) można podać między innymi stałą `os.WNOHANG`, która sprawia, że odpowiednia funkcja nie czeka rzeczywiście na zakończenie jakiegoś potomka, lecz zwraca status jednego z już zakończonych — albo odpowiednią krotkę wypełnioną zerami, gdy żaden proces nie czeka w kolejce zakończonych.

Na listingu A.3 widzimy schemat serwera współbieżnego wieloprocesowego opartego o funkcję `os.fork` (por. listing 8.1 na str. 103) natomiast listing A.4 pokazuje, jak z programu w Pythonie zrobić demona (por. listing 8.3 na str. 106).

Listing A.3. Schemat serwera współbieżnego wieloprocesowego w Pythonie

```
import socket
import os

...

gn0 = socket.socket(...)
gn0.bind(...)
gn0.listen(...)

while True:
    pol = gn0.accept()
    if os.fork() != 0:
        pol.close()
    else:
        gn0.close()
        # tu obsługa gniazda pol

...
```

Listing A.4. Algorytm stworzenia Uniksowego demona w Pythonie

```
import os
import sys
```

```
4 def fork_exit():
      try:
6         pid = os.fork()
      except OSError, e:
8         print >> sys.stderr, "Fork: %s [%s]." % (
              e.strerror, e.errno)
10        os._exit(1)
      if (pid != 0):
12        os._exit(0)

14 fork_exit()

16 os.setsid()

18 fork_exit()

20 os.chdir("/")
   os.umask(0)
22
   os.close(0)
24 os.close(1)
   os.close(2)
26
   os.open("/dev/null", os.O_RDWR)
28
   os.dup2(0, 1)
30 os.dup2(0, 2)

32 # tu już właściwy program...
```

Z modułu `select` zaprezentujemy jedną tylko funkcję — o nazwie `select`:

```
select.select(rlist, wlist, xlist[, timeout])
    # zwraca (rready, wready, xready)
```

Działanie tej funkcji jest właściwie identyczne z odpowiednią funkcją systemową, jednakże trzy pierwsze argumenty są po prostu Pythonowymi listami — i to niekoniecznie deskryptorów (choć i te mogą tam się znajdować), ale także dowolnych obiektów „plikopodobnych", które mają metodę `fileno` zwracającą odpowiedni deskryptor pliku. Mogą to być więc obiekty plików, ale także gniazd, standardowych strumieni itp.). Opuszczenie ostatniego parametru (normalnie jest to zmiennoprzecinkowa liczba sekund) oznacza oczekiwanie blokujące (w nieskończoność lub do odpowiedniej zmiany stanu

któregoś z obiektów w trzech pierwszych parametrach), a `timeout==0` oznacza brak oczekiwania (tylko sprawdzenie gotowości odpowiednich obiektów).

Natomiast wartością zwracaną jest w tej funkcji krotka trzech list, które zawierają te elementy (spośród podanych w trzech pierwszych parametrach), które są gotowe (odpowiednio: do odczytu, zapisu oraz sprawdzenia sytuacji wyjątkowej).

Na listingu A.5 widzimy schemat serwera obsługującego wiele połączeń, ale opartego o funkcję `select.select` (por. listing 8.2 na str. 104).

Listing A.5. Schemat serwera jednoprocesowego wielopołączeniowego w Pythonie

```
import socket
import select

...

gn0 = socket.socket(...)
gn0.bind(...)
gn0.listen(...)

rl = [gn0]

while True:
    rr, wr, xr = select.select(rl, [], [])
    for gn in rr:
        if gn is gn0:
            pol = gn0.accept()
            rl += [pol]
        else:
            # obsługa pojedynczego czytania z gn
            # ewentualne jego zamknięcie
            # wraz z usunięciem z listy rl

...
```

W końcu w module `signal` mogą przydać się stałe oznaczające poszczególne sygnały (znajdziemy tu między innymi `signal.SIGHUP`, `signal.SIGKILL` oraz `signal.SIGTERM`), które mogą być użyte z wspomnianą wyżej funkcją `os.kill`, a także z poniższymi.

```
signal.getsignal(signalnum)
signal.signal(signalnum, handler)
    # obie zwracają dotychczasowy handler
    # danego sygnału
```

Pierwsza z tych funkcji zwraca po prostu Pythonową funkcję obsługująca sygnał (lub jedną ze stałych: `signal.SIG_IGN`, gdy sygnał jest ignorowany; `signal.SIG_DFL`, gdy sygnał jest obsługiwany domyślnie; `None`, gdy obsługa nie była zainstalowana z poziomu Pythona). Druga także zwraca (w ten sam sposób) dotychczasowy sposób obsługi danego sygnału, ale zmienia go także na nowy — `handler` musi tu być jedną ze stałych `signal.SIG_IGN` albo `signal.SIG_DFL` lub też Pythonową funkcją przyjmującą dwa parametry (pierwszy z nich jest numerem sygnału, drugi to obiekt aktualnej ramki Pythonowego stosu). Wspominamy tutaj o tym, warto bowiem (jak to sygnalizowaliśmy w rozdziale 8) w serwerach/demonach przewidzieć co najmniej obsługę sygnałów `signal.SIGHUP` (do ponownego wczytania konfiguracji i „miękkiego" restartu serwera) oraz `signal.SIGTERM` (do „eleganckiego" zamknięcia serwera).

Listing A.6 pokazuje schemat zapewnienia odpowiedniej obsługi przykładowego sygnału.

Listing A.6. Obsługa sygnału w Pythonie

```
import signal
import os

def miekki_restart(signum, frame):
    signal.signal(signum, signal.SIG_IGN)
    # tu czynności restartujące
    signal.signal(signum, miekki_restart)

def lagodny_koniec(signum, frame):
    signal.signal(signum, signal.SIG_IGN)
    # tu czynności kończące
    os._exit(0)

signal.signal(signal.SIGHUP, miekki_restart)
signal.signal(signal.SIGTERM, lagodny_koniec)

# tu ciąg dalszy programu...
```

A.3. Moduły wyższego poziomu

Silną stroną Pythona jest rozbudowana biblioteka standardowa, która zawiera wiele modułów, klas i funkcji bardzo wysokiego poziomu. Wybrane przedstawimy tutaj na zakończenie rozdziału.

A.3.1. Sieciowe elementy modułu `multiprocessing`

Moduł[5] `multiprocessing` jest obszernym zbiorem klas pozwalającym na przenośne programowanie współbieżne i rozproszone. W związku z tym, oferuje także podmoduł `multiprocessing.connection`, związany z klasą `multiprocessing.Connection` będącą wyższą warstwą abstrakcji nad gniazdami sieciowymi[6]. Filozofia samego połączenia jest analogiczna do połączeń na poziomie gniazd TCP, jednakże tutaj odpowiednie obiekty klasy `multiprocessing.Connection` tworzone są pojedynczymi konstruktorami (`multiprocessing.connection.Listener` dla serwera oraz `multiprocessing.connection.Client` dla klienta), które dodatkowo oferują uwierzytelnianie. Metody obsługujące tego rodzaju obiekty są analogiczne do metod gniazdowych, jednakże umożliwiają bezpośrednie przesyłanie różnych „piklowalnych" [65] obiektów Pythonowych. Niżej przedstawiamy kilka takich funkcji (przy założeniu, że `c` jest obiektem klasy `multiprocessing.Connection`.

```
c.accept()
    # zwraca nowy obiekt połączony
c.close()
    # zamyka połączenie
c.last_accepted
    # przechowuje adres ostatnio zaakceptowanego
    # połączenia
c.send(obj)
    # wysyła piklowalny obiekt i zwraca None
c.recv()
    # zwraca obiekt odebrany z połączenia
c.send_bytes(buffer[, offset[, size]])
    # wysyła ciąg bajtów i zwraca None
c.recv_bytes([maxlength])
    # odbiera ciąg bajtów
c.fileno()
    # zwraca odpowiedni deskryptor pliku
    # (np. dla select.select)
```

Poniższe dwa listingi ukazują omawiane tutaj obiekty w użyciu. Przy okazji, w listingu A.7 widzimy konstrukcję serwera współbieżnego wieloprocesowego przy pomocy przenośnego modułu `multiprocessing` zamiast funkcji `fork`.

[5] Właściwie *pakiet* w nomenklaturze Pythona, bo w jego skład wchodzą także inne moduły.

[6] A także gniazdami lokalnymi systemu Unix oraz łączami nazwanymi systemu MS Windows.

Listing A.7. Serwer echa

```
import multiprocessing.connection as Co
import multiprocessing as MP
import time

l = Co.Listener(address=('', 54391), authkey='haslo')

def obsluga(pol, adr):
    print MP.current_process(), "Laczy␣sie:␣", adr
    ob = pol.recv()
    time.sleep(3)
    print ob
    pol.send(ob)
    pol.close()

while True:
    c = l.accept()
    MP.Process(target=obsluga,
        args=(c, l.last_accepted)).start()
```

Listing A.8. Klient echa

```
import multiprocessing.connection as Co

c = Co.Client(address=('matrix.umcs.pl', 54391),
                authkey='haslo')

oryg = [1, 2.5, "koza"]
c.send(oryg)
ob = c.recv_bytes()
c.close()
print ob, oryg
#ob[1] = 10000000000
#print ob, oryg
```

A.3.2. Obiekty rozproszone w module `multiprocessing`

Moduł `multiprocessing` wprowadza także obiekty rozproszone, które kontaktują się ze sobą za pomocą menedżerów obiektów rozproszonych obsługiwanych przez podmoduł `multiprocessing.managers`. Poniższe listingi powinny pokazać czytelne przykłady użycia menedżerów do programowania sieciowego. Pierwsze trzy (A.9–A.11) stanowią proste wprowadzenie do obiektów rozproszonych i ich menedżerów, natomiast listingi A.12–A.13 po-

kazują bardziej praktyczne ich zastosowanie — do implementacji pogawędki przez sieć.

Listing A.9. Przykład prostego serwera obiektów rozproszonych

```
import multiprocessing as mp
import multiprocessing.managers as man
import time

class Obliczenia(object):
    def set(self, x):
        self.w = x
        print "SET", x
        print repr(self)
    def get(self):
        print "GET", self.w
        return self.w
    def plus(self, y):
        return self.w+y
    def razy(self, y):
        return self.w*y

class MojManager(man.BaseManager):
    pass

MojManager.register("Obl", Obliczenia)

m = MojManager(address=('', 51239), authkey="klucz")

m.get_server().serve_forever()
```

Listing A.10. Przykład prostego klienta obiektów rozproszonych

```
import multiprocessing as mp
import multiprocessing.managers as man
import time

class LokManager(man.BaseManager):
    pass

LokManager.register("Obl")

m = LokManager(address=('matrix.umcs.pl', 51239),
                authkey="klucz")

m.connect()
```

```
o1 = m.Obl()
o1.set(210)
time.sleep(2)
print o1.get()
print o1.dane
```

Listing A.11. Przykład nieco innego serwera obiektów rozproszonych działającego z tym samym klientem

```
import multiprocessing as mp
import multiprocessing.managers as man
import time

class Obliczenia(object):
    dane = []
    def set(self, x):
        self.dane.append(x)
        print "SET", x
    def get(self):
        print "GET", self.dane
        return self.dane
#    def plus(self, y):
#        return self.w+y
#    def razy(self, y):
#        return self.w*y

class MojManager(man.BaseManager):
    pass

MojManager.register("Obl",
    Obliczenia)

m = MojManager(address=('', 51239),
                authkey="klucz")

m.get_server().serve_forever()
```

Listing A.12. Serwer rozmowy

```
import multiprocessing as mp
import multiprocessing.managers as man
import time

class Rozmowa(object):
    kolejki = {}
```

```
7       def start(self, nazwa):
            print "START:", nazwa
9           self.kolejki[nazwa] = mp.Queue()
            print "PO␣START:", nazwa
11      def napisz(self, nazwa, tresc):
            for kto, kol in self.kolejki.items():
13              if kto!=nazwa:
                    print "PUT:", nazwa
15                  kol.put((nazwa, tresc))
                    print "PO␣PUT:", nazwa
17          print nazwa, ":", tresc
        def odbierz(self, nazwa):
19          print "GET:", nazwa
            mess = self.kolejki[nazwa].get()
21          print "PO␣GET:", nazwa, mess
            return mess
23
   class MojManager(man.BaseManager):
25      pass

27 MojManager.register("R", Rozmowa)

29 m = MojManager(address=('', 51739),
                  authkey="klucz")
31
   m.get_server().serve_forever()
```

Listing A.13. Klient rozmowy

```
1  import multiprocessing as mp
   import multiprocessing.managers as man
3  import time

5  class LokManager(man.BaseManager):
        pass
7
   LokManager.register("R")
9
   m = LokManager(address=('matrix.umcs.pl', 51739),
11                  authkey="klucz")

13 m.connect()

15 imie = raw_input("Imie?␣")

17 r = m.R()
```

```
    r.start(imie)
19
    def czyt(r):
21      while True:
            print r.odbierz(imie)
23
    p1 = mp.Process(target=czyt, args=(r,))
25  p1.start()

27  while True:
        r.napisz(imie, raw_input("> "))
```

A.3.3. Przegląd innych modułów sieciowych

Na zakończenie tego dodatku wymienimy niektóre z innych modułów sieciowych dostępnych standardowo w Pythonie.

Moduł `SocketServer` oferuje klasy służące do implementacji serwerów gniazd, na których to klasach można budować — abstrahując od technicznych szczegółów — własne klasy do obsługi protokołów warstwy aplikacji.

Moduł `httplib` implementuje protokół HTTP po stronie klienta.

Moduł `ftplib` implementuje protokół FTP po stronie klienta.

Moduł `poplib` implementuje protokół POP3 po stronie klienta.

Moduł `imaplib` implementuje protokół IMAP4 po stronie klienta.

Moduł `nntplib` implementuje protokół NNTP po stronie klienta.

Moduł `smtplib` implementuje protokół SMTP po stronie klienta.

Moduł `smtpd` implementuje protokół SMTP po stronie serwera.

Moduł `telnetlib` implementuje protokół Telnet po stronie klienta.

Moduły `BaseHTTPServer`, `SimpleHTTPServer`, `CGIHTTPServer` implementują serwery protokołu HTTP na różnym poziomie skomplikowania.

Dodatek B

Jak to jest w Javie?

B.1. Adresy, numery portów a nazwy usług, kolejność bajtów **166**
B.1.1. Adresy 166
B.1.2. Numery portów a nazwy usług 168
B.1.3. Kolejność bajtów 168
B.2. Gniazda TCP **169**
B.2.1. Tworzenie gniazda i nawiązywanie połączenia . 169
B.2.2. Przesyłanie danych przez gniazdo 170
B.2.3. Tworzenie gniazda nasłuchującego i akceptowanie połączeń 171
B.3. Gniazda UDP **171**
B.3.1. Zgłaszanie błędów 172
B.3.2. Opcje gniazd 173
B.4. Serwery współbieżne **173**
B.5. Zaawansowane operacje z użyciem `java.nio` **174**
B.5.1. Nieblokujące wejście-wyjście 174
B.5.2. Zwielokrotnianie wejścia-wyjścia 176
B.6. Protokoły warstwy aplikacji **177**

Niniejszy dodatek poświęcony jest programowaniu warstwy aplikacji w języku Java. Skupimy się na zaprezentowaniu umożliwiających to elementów biblioteki standardowej Javy. Nie będziemy omawiać szczegółowo wszystkich używanych klas i metod. Skupimy się na wybranych. W razie potrzeby zawsze można sięgnąć do oficjalnej dokumentacji biblioteki. Pełna dokumentacja najnowszej wersji Javy (w chwili pisania tej książki jest to wersja 1.7.3) dostępna jest obecnie na stronie `http://docs.oracle.com/javase/7/docs/api/`.

Programowaniu sieciowemu w Javie poświęcone są w całości m.in. książki [23] i [44].

Java jest językiem obiektowym i — w przeciwieństwie np. do Pythona — wymuszającym programowanie w stylu obiektowym. Ma to duży wpływ na jej bibliotekę standardową. O ile w przypadku Pythona wiele metod dotyczących interfejsu gniazd jest po prostu opakowaniem pojedynczego wywołania funkcji z języka C w typy Pythona, a odpowiednikiem deskryptora gniazda jest po prostu obiekt `socket`, to interfejs gniazd w Javie zdecydowanie mniej przypomina ten w C.

W Javie mamy zaprojektowaną całą hierarchię klas związaną z programowaniem interfejsu gniazd. Metody tych klas w niewielkim stopniu odpowiadają funkcjom interfejsu gniazd z języka C.

Klasy związane bezpośrednio z programowaniem sieciowym w Javie znajdują się w pakiecie `java.net`.

Interfejs gniazd w standardowej bibliotece Javy udostępnia tylko interfejs do protokołów TCP i UDP. Nie ma możliwości korzystania z gniazd surowych i programowania bezpośrednio na poziomie datagramów IP lub komunikatów ICMP[1].

B.1. Adresy, numery portów a nazwy usług, kolejność bajtów

B.1.1. Adresy

Wszystkie funkcje wymagające adresu zadanego jako napis akceptują zarówno adresy IP (w postaci z kropkami) jak i nazwy domenowe. Ewentualne przekształcenia adresów (i odpytywanie serwerów DNS) wykonywane jest automatycznie.

Klasą reprezentującą w Javie adres IP jest `InetAddress`. Obiekty tej klasy używane są jako argumenty funkcji wymagających adresów IP. Kla-

[1] Z wyjątkiem JNI (ang. *Java Native Interface*) — mechanizmu umożliwiającego używanie bibliotek napisanych w innych językach takich jak C czy C++.

sa ta odpowiada także za przekształcenia między adresami IP a adresami domenowymi.

Obiekty tej klasy można tworzyć za pomocą kilku statycznych metod. Argumenty zadane napisami mogą być, jak to było wyżej powiedziane, zarówno adresami IP jak i adresami domenowymi. W razie potrzeby dokonywana jest automatyczna konwersja na adres IP. Wybranymi metodami służącymi do tworzenia obiektów tej klasy są:

— `static InetAddress getByName(String host)` — zadana nazwa domenowa lub tekstowa reprezentacja adresu IP,
— `static InetAddress[] getAllByName(String host)` — zwraca tablicę wszystkich adresów zadanego hosta,
— `static InetAddress getByAddress(byte[] addr)` — adres IP zadany jako tablica bajtów,
— `static InetAddress getLocalHost()` — zwraca adres sieciowy lokalnego hosta.
— `static InetAddress getLoopbackAddress()` — zwraca adres pętli zwrotnej (patrz podrozdział 2.2.1.2).

Innymi przydatnymi metodami tej klasy są:

— `String getHostName()` — zwraca nazwę domenową,
— `String getHostAddress()` — zwraca tekstową reprezentację adresu IP,
— `String getAddress()` — zwraca adres IP jako tablicę bajtów,
— `String getCanonicalHostName()` — zwraca pełną nazwę domenową dla danego hosta.

Metody te w razie potrzeby także dokonują automatycznego przekształcenia adresu.

Klasami dziedziczącymi po ogólnej klasie `InetAddress` reprezentującymi adresy IPv4 i IPv6 są klasy `Inet4Address` i `Inet6Address`. Zazwyczaj nie ma potrzeby korzystania bezpośrednio z tych klas.

Listing B.1 prezentujący użycie tej klasy jest odpowiednikiem programów z listingów 3.13 i 3.14.

Listing B.1. Użycie metody `getAllByName` z klasy`InetAddress`

```
import java.net.*;
2
public class Main {
4
  public static void main(String[] args) {
6   if (args.length < 1) {
      System.err.println("Brak␣argumentu.");
8     System.exit(1);
    }
```

```
10      try {
          for (InetAddress inetAddress :
12                Inet4Address.getAllByName(args[0])) {
            System.out.println(
14                  inetAddress.getHostAddress());
          }
16      } catch (UnknownHostException ex) {
          System.err.println("Błąd");
18        System.exit(1);
        }
20    }
    }
```

Klasą reprezentującą adres gniazda (parę adres IP + numer portu) jest `InetSocketAddress` dziedzicząca po abstrakcyjnej klasie `InetAddress`. Konstruktory tej klasy otrzymują adres hosta i numer portu. Klasa ta ma metody takie jak `getAddress` i `getPort` zwracające odpowiednio adres hosta i numer portu zadanego adresu gniazda.

B.1.2. Numery portów a nazwy usług

W standardowej bibliotece Javy nie ma metod wykonujących zadania funkcji `getservbyname` i `getservbyport`, nie ma więc przenośnej metody odwzorowania nazw usług na numery portów.

B.1.3. Kolejność bajtów

Klasy i metody Javy nie dają nam bezpośredniego dostępu do struktur adresowych takich jak w języku C. Nie musimy zajmować się takimi rzeczami jak rozmiary tych adresów, czy kolejność bajtów w nich. Operujemy tutaj na wyższym poziomie abstrakcji — mamy klasy i metody zajmujące się tymi zagadnieniami i udostępniające interfejs dużo prostszy niż oryginalny interfejs gniazd dostępny w języku C.

Kolejnym powodem, dla którego w Javie dużo rzadziej musimy się zajmować kolejnością bajtów jest to, że Java przechowuje wewnętrznie wszystkie dane liczbowe w formacie big-endian, czyli takim samym jak standardowa sieciowa kolejność bajtów.

Jeżeli jednak musimy zajmować się kolejnością bajtów (bo wiemy, że np. otrzymane dane są w formacie little-endian), to możemy użyć klasy `ByteBuffer` z pakietu `java.nio` umożliwiającej określenie kolejności bajtów w danych czytanych i zapisywanych do bufora[2].

[2] Za pomocą argumentów typu `ByteOrder` mogących mieć wartość `ByteOrder.BIG_ENDIAN` lub `ByteOrder.LITTLE_ENDIAN`.

B.2. Gniazda TCP

Klasami reprezentującymi gniazda TCP są klasy `Socket` i `ServerSocket`. `Socket` reprezentuje gniazda służące do bezpośredniej komunikacji — gniazda klienta i gniazda serwera zwrócone przez metodę `accept` akceptującą połączenia. `ServerSocket` jest klasą reprezentującą gniazdo nasłuchujące TCP.

B.2.1. Tworzenie gniazda i nawiązywanie połączenia

Konstruktory klasy `Socket` takie jak:

— `Socket(InetAddress address, int port)`
— `Socket(String host, int port)`

umożliwiają nawiązanie od razu połączenia z zadanym argumentami zdalnym gniazdem. Konstruktor `Socket()` tworzy niepołączone gniazdo. Metodą umożliwiającą połączenie jest `void connect(SocketAddress endpoint)`. Sprawdzić, czy gniazdo jest połączone, możemy metodą `bool isConnected()`. Metoda `void shutdownOutput()` zamyka gniazdo do zapisu (odpowiednik wywołania funkcji `socket` z argumentem `SHUT_WR`) w języku C. Jest też analogiczna funkcja `shutdownInput`. Metoda `void close()` zamyka gniazdo, a `bool isClosed()` sprawdza, czy gniazdo jest zamknięte.

Wymienione poniżej metody zwracają informacje o obu stronach nawiązanego połączenia.

Informację o lokalnej części gniazda (odpowiednik `getsockname` z C): zwracają metody:

— `InetAddress getLocalAddress()` — zwraca adres IP, do którego gniazdo jest lokalnie dowiązane,
— `int getLocalPort()` — zwraca port, do którego dowiązane jest gniazdo,
— `SocketAddress getLocalSocketAddress()` — zwraca lokalny adres (adres IP + numer portu), do którego dowiązane jest gniazdo,

Informację o zdalnej części gniazda (odpowiednik `getpeername` z C): zwracają metody:

— `InetAddress getInetAddress()` — zwraca adres IP drugiej strony połączenia,
— `int getPort()` — zwraca port drugiej strony połączenia,
— `SocketAddress getRemoteSocketAddress()` — zwraca adres drugiej strony połączenia.

B.2.2. Przesyłanie danych przez gniazdo

Przesyłanie danych za pomocą gniazd odbywa się za pomocą strumieni. Możemy w związku z tym w łatwy sposób korzystać ze wszystkich oferowanych przez strumienie udogodnień, np. strumienie znakowe z określonym kodowaniem czy podział na wiersze.

Odpowiednie strumienie uzyskujemy za pomocą dwóch metod:

— `InputStream getInputStream()` — zwraca obiekt typu `InputStream` służący do odbierania danych z gniazda,
— `OutputStream getOutputStream()` — zwraca obiekt typu `OutputStream` służący do wysyłania danych do gniazda.

Należy zawsze pamiętać, żeby nie zamykać gniazda przed zamknięciem związanych z nim strumieni. Może to spowodować np. w przypadku buforowanych strumieni niewysłanie danych znajdujących się w buforach. Zamknięcie strumienia powoduje także zamknięcie powiązanego z nim gniazda.

Na przykład fragment kodu na listingu B.2 czyta z gniazda `sock` odsyła odpowiedzi (określone gdzieś metodą `response`) na jednowierszowe żądania. Zarówno żądania jak i odpowiedzi kodowane są za pomocą UTF-8.

Listing B.2. Użycie strumieni

```
   InputStream in = sock.getInputStream();
 2 Reader reader = new InputStreamReader(in,
         StandardCharsets.UTF_8);
 4 BufferedReader bufReader =
         new BufferedReader(reader);
 6
   OutputStream out = sock.getOutputStream();
 8 Writer writer = new OutputStreamWriter(out,
         StandardCharsets.UTF_8);
10 BufferedWriter bufWriter = new BufferedWriter(
         writer);
12
   String line;
14 while ((line = bufReader.readLine()) != null) {
     bufWriter.write(response(line) + "\r\n");
16   bufWriter.flush();
   }
18
   bufReader.close();
20 bufWriter.close();
```

B.2.3. Tworzenie gniazda nasłuchującego i akceptowanie połączeń

Konstruktory

— `ServerSocket(int port)`
— `ServerSocket(int port, int backlog)`

tworzą gniazda od razu dowiązane do odpowiedniego portu i nasłuchujące. Argument `backlog` jest tym samym co argument do funkcji `listen` z interfejsu gniazd języka C[3]. Konstruktor `ServerSocket()` tworzy gniazdo niedowiązane. Dowiązać takie gniazdo można za pomocą metody `void bind(SocketAddress endpoint)`. Metoda `bool isBound()` sprawdza, czy gniazdo jest dowiązane.

Metoda `Socket accept()` akceptuje połączenie i zwraca nowe gniazdo służące do obsługi klienta.

Metoda `void close()` zamyka gniazdo.

Listing B.3 prezentuje fragment iteracyjnego serwera wysyłającego każdemu klientowi pozdrowienia i kończącego połączenie:

Listing B.3. Użycie klasy `ServerSocket`

```
  ServerSocket serv = new ServerSocket(port);
2 while (true) {
    Socket client = serv.accept();
4   PrintStream p = new PrintStream(
            client.getOutputStream());
6   p.println("Pozdrowienia");
    p.close();
8   client.close();
  }
```

B.3. Gniazda UDP

Gniazda UDP reprezentowane są przez klasę `DatagramSocket`. Datagramy reprezentowane są przez klasę `DatagramPacket`. Podstawową różnicą między tą klasą, a klasą `Socket` jest brak możliwości uzyskania strumienia związanego z gniazdem (bo UDP nie jest strumieniowy) i obecność metod:

— `void send(DatagramPacket p)` — wysyła datagram,
— `void receive(DatagramPacket p)` — odbiera datagram.

[3] W Javie nie ma osobnej funkcji `listen`, dowiązanie gniazda `ServerSocket` do portu tworzy od razu gniazdo nasłuchujące.

Z każdym obiektem reprezentującym datagram związane są informacje o danych przesyłanych przez ten datagram oraz o adresie gniazda (adres IP + port). Adres gniazda w pakiecie odebranym oznacza adres nadawcy pakietu, a w pakiecie wysyłanym adres musi wskazywać na odbiorcę pakietu. Fragment kodu na listingu B.4 odpowiada za wysłanie krótkiego napisu pod adres zadany zmiennymi `host` i `port` oraz odebranie odpowiedzi do zmiennej napisowej `resp`. Kod na listingu B.5 odbiera datagram, wyświetla adres nadawcy i odsyła datagram w niezmienionej postaci. `BUF_SIZE` w obu listingach musi być pewną stałą określającą rozmiar bufora.

Listing B.4. Klient UDP

```
   DatagramSocket sock = new DatagramSocket();
 2 byte[] sendBuf = msg.getBytes();
   byte[] recvBuf = new byte[BUF_SIZE];
 4 DatagramPacket packet = new DatagramPacket(
           sendBuf, sendBuf.length,
 6         new InetSocketAddress(host, port));
   sock.send(packet);
 8 packet.setData(recvBuf);
   sock.receive(packet);
10 resp = new String(packet.getData(), 0,
           packet.getLength());
```

Listing B.5. Serwer UDP

```
1 DatagramSocket sock = new DatagramSocket(port);
  byte[] buf = new byte[BUF_SIZE];
3 DatagramPacket packet = new DatagramPacket(buf,
          BUF_SIZE);
5 sock.receive(packet);
  System.out.println(
7         packet.getAddress().getHostName()
          + ":" + packet.getPort());
9 sock.send(packet);
```

B.3.1. Zgłaszanie błędów

Błędy związane z gniazdami są w Javie zgłaszane za pomocą wyjątków klas pochodnych klasy `SocketException`. Są to m.in. takie wyjątki jak `BindException` — brak możliwości dowiązania gniazda do lokalnego adresu, czy `ConnectException` — brak możliwości połączenia ze zdalnym adresem.

B.3.2. Opcje gniazd

Każdej opcji gniazd obsługiwanej przez bibliotekę standardową Javy odpowiada para metod `get...` i `set...`.

Przykładami są

— `void setReceiveBufferSize(int size)` i `int getReceiveBufferSize()` dla opcji `SO_RCVBUF`,
— `void setSendBufferSize(int size)` i `int getSendBufferSize()` dla opcji `SO_SNDBUF`,
— `void setReuseAddress(boolean on)` i `boolean getReuseAddress()` dla opcji `SO_REUSEADDR`,
— `void setTcpNoDelay(boolean on)` i `boolean getTcpNoDelay()` dla opcji `TCP_NODELAY`,

Pozostałe można znaleźć w dokumentacji klas `Socket` i `DatagramSocket`.

B.4. Serwery współbieżne

Podstawowym mechanizmem oferującym współbieżność w Javie są wątki. Na listingu B.6 pokazany jest uproszczony (bez obsługi błędów) schemat prostego serwera współbieżnego zrealizowanego za ich pomocą.

Listing B.6. Serwer współbieżny

```
public class Server {

  static class ServeClient implements Runnable {
    private Socket client;
    public ServeClient(Socket client) {
      this.client = client;
    }
    public void serveClient(Socket client) {
        ... // obsługa klienta
    }
    public void run() {
      /* obsługa gniazda klienta (client)  */
      serveClient(client);
    }
  }

  public static void main(String[] args)
          throws Exception {
    echoServer(5678);
  }
```

```
22    public static void server(int port) {
        ServerSocket serv = new ServerSocket(port);
24      while (true) {
          Socket client = serv.accept();
26        new Thread(new ServeClient(client)).start();
        }
28    }

30  }
```

B.5. Zaawansowane operacje z użyciem `java.nio`

Oprócz trzech omówionych klas reprezentujących gniazda — `Socket`, `ServerSocket` i `DatagramSocket` — są też w pakiecie `java.nio` trzy odpowiadające im klasy: `SocketChannel`, `ServerSocketChannel` i `DatagramChannel`. Klasy te umożliwiają zaawansowane operacje niemożliwe do wykonania za pomocą wcześniejszych klas — nieblokujące operacje wejścia-wyjścia oraz zwielokrotnianie wejścia-wyjścia[4].

B.5.1. Nieblokujące wejście-wyjście

Jeżeli chcemy mieć możliwość nieblokującego wykonywania operacji takich jak `open`, `accept`, `read` i `write` na gniazdach TCP, to musimy utworzyć gniazdo korzystając z klasy `SocketChannel` lub `ServerSocketChannel` dla gniazda nasłuchującego.

Schemat utworzenia gniazda klienta i korzystania z nieblokujących operacji przedstawiony jest na listingu B.7. Pominięta została obsługa błędów. Klient wysyła żądanie i odbiera odpowiedź. Zakładamy, że serwer zamyka połączenie po odesłaniu odpowiedzi.

Listing B.7. Schemat klienta korzystającego z gniazd nieblokujących

```
    static final int BUF_SIZE = 1024;
2
    void nonblockingClient(String host, int port,
4           String request) {
      SocketChannel channel = SocketChannel.open();
6     channel.configureBlocking(false);

8     channel.connect(new InetSocketAddress(host,
              port));
10    while (!channel.finishConnect()) {
```

[4] Analogiczne do funkcji `select` opisanej w podrozdziale 8.3.

```
      // oczekujemy na połączenie
12    // robimy chwilowo coś innego
      ...
14  }

16  ByteBuffer sendBuf = ByteBuffer.wrap(
            request.getBytes());
18  System.out.println(sendBuf.remaining());
    while (sendBuf.hasRemaining()) {
20    int nWrite = channel.write(sendBuf);
      if (nWrite == 0) {
22      // wysyłanie nie jest na razie możliwe
        // wykonujemy inne czynności
24      ...
      }
26  }

28  ByteBuffer recvBuf = ByteBuffer.allocate(
            BUF_SIZE);
30  int nRead = channel.read(recvBuf);
    while (nRead != -1) \\ -1 oznacza koniec danych
32  {
      if (nRead == 0) {
34      // czekamy na dane
        // wykonujemy inne czynności
36      ...
      }
38    nRead = channel.read(recvBuf);
    }
40
    channel.close();
42 }
```

Obiekty typu `SocketChannel` lub `ServerSocketChannel` tworzymy korzystając ze statycznych metod `open` (wiersz nr 5). Tryb blokowania ustawiamy metodą `configureBlocking` (wiersz nr 6). Domyślny jest tryb blokujący — odpowiada parametrowi `true`. Metoda `finishConnect` informuje, czy połączenie jest już nawiązane. Jeżeli nie — możemy się chwilowo zająć czymś innym (wiersze 9–13). Taki schemat występuje też dla metod `read` i `write`. Jako argumenty metod `read` i `write` przekazywane są bufory typu `ByteBuffer`. Informacje o nich można znaleźć w dokumentacji biblioteki.

W analogicznym nieblokującym wykorzystaniu gniazda nasłuchującego metoda `accept` zwraca gniazdo połączone (jako `SocketChannel`) lub `null`, jeżeli nikt nie oczekuje na połączenie.

B.5.2. Zwielokrotnianie wejścia-wyjścia

Mechanizmem analogicznym do funkcji `select` jest w Javie klasa `Selector`. Jej użycie pokazane jest na listingu B.8. Tutaj także pominięta jest obsługa błędów.

Listing B.8. Serwer echa korzystający z klasy `Selector`

```
static final int BUF_SIZE = 1024;
ByteBuffer buf = ByteBuffer.allocate(BUF_SIZE);

void echo(int port) {
  ServerSocketChannel server =
          ServerSocketChannel.open();
  server.configureBlocking(false);
  server.socket().bind(new InetSocketAddress(port));

  Selector selector = Selector.open();
  server.register(selector, SelectionKey.OP_ACCEPT);

  while (true) {
    selector.select();
    for (SelectionKey key :
            selector.selectedKeys()) {
      if (key.isValid()) {
        if (key.isAcceptable()) {

          ServerSocketChannel serv =
                  (ServerSocketChannel) key.channel();
          SocketChannel cli = serv.accept();
          if (cli != null) {
            cli.configureBlocking(false);
            cli.register(selector,
                    SelectionKey.OP_READ);
          }

        } else if (key.isReadable()) {

          SocketChannel cli =
                  (SocketChannel) key.channel();
          if (cli.read(buf) == -1) {
            key.cancel();
            cli.close();
          } else {
            buf.flip();
            while (buf.hasRemaining()) {
              cli.write(buf);
            }
```

```
                buf.clear();
42
            }
44        }
        }
46    }
    }
48 }
```

Wszystkie gniazda, dla których chcemy oczekiwać na jakieś zdarzenie (możliwość akceptacji połączenia, czytania danych lub pisania) muszą być nieblokujące. Gniazdo reprezentowane przez jeden z typów `SocketChannel` lub `ServerSocketChannel` musi się zarejestrować w selektorze metodą `register` (wiersze 11 i 26) z odpowiednim typem zdarzenia (np. `OP_ACCEPT` — wiersz 11 lub `OP_READ` — wiersz 26). Metoda `select` działa analogicznie do funkcji `select` znanej z C — kończy działanie zaznaczając, na których gniazdach możemy wykonać jaką operację. Tutaj możemy to odczytać metodą `selectedKeys` (wiersz 16) zwracającą zbiór „kluczy" selektora — każdy odpowiada gniazdu — oraz metodami `isAccceptable` i `isReadable` wywołanymi dla kluczy. Jeżeli gniazdo zamknęło połączenie (rozpoznajemy to w wierszu 33), to usuwamy klucz z selektora (metoda `cancel` — wiersz 34).

B.6. Protokoły warstwy aplikacji

Klasami umożliwiającymi korzystanie z zasobów dostępnych za pomocą URL-i są klasy `URL`, `URLConnection` i `HttpURLConnection`.

Klasą reprezentującą URL w Javie jest klasa `URL`. Do skonstruowania obiektu tej klasy możemy użyć jednego z kilku konstruktorów w zależności od tego, które części URL-a są niepuste. Przykładowymi konstruktorami są

- `URL(String spec)` — np. `new URL("http://example.com:8080/plik.pdf")`, może zgłosić wyjątek `MalformedURLException` jeżeli URL nie ma poprawnego formatu.
- `URL(String protocol, String host, int port, String file)` — np.`new URL("http", "example.com", 8080, "/plik.pdf")`.

Klasa ta ma metody umożliwiające rozbiór URL-a na części, np.:

- `String getFile()`
- `String getHost()`
- `String getPath()`
- `int getPort()`
- `String getProtocol()`

— `String getQuery()`

Jednak najważniejszą funkcją tej klasy jest możliwość łatwego pobierania zasobów dostępnych przez URL. Jeżeli nie interesują nas w ogóle szczegóły protokołu używanego do pobrania zasobu, to nie jest to trudniejsze od skopiowania lokalnego pliku. Możliwość taką daje metoda `InputStream openStream()`. Otwiera ona strumień bajtowy, za pomocą którego możemy przeczytać zawartość zasobu.

Przykład takiego użycia tej klasy przedstawiony jest na listingu B.9 (bez obsługi błędów).

Listing B.9. Użycie klasy `URL` do pobrania pliku

```
public static final int BUF_SIZE = 4096;
void pobierzPlik(String Url, String fileName) {
  URL url = new URL(Url);
  InputStream urlInput = url.openStream();
  FileOutputStream fileOutput
          = new FileOutputStream(fileName);
  byte [] buf = new byte[BUF_SIZE];
  int nBytes = urlInput.read(buf);
  while (nBytes != -1) {
    fileOutput.write(buf, 0, nBytes);
    nBytes = urlInput.read(buf);
  }
  urlInput.close();
  fileOutput.close();
}
```

Metoda `URLConnection openConnection()` zwraca obiekt klasy `URLConnection` umożliwiający dostęp do niektórych szczegółów związanych z protokołem. W przypadku protokołu HTTP tak naprawdę zwrócony będzie obiekt klasy `HttpURLConnection` dziedziczącej po `URLConnection`, np.

```
URL url = new URL(
        "http://www.gnu.org/licenses/gpl.txt");
HttpURLConnection urlConn =
        (HttpURLConnection) url.openConnection();
```

Korzystając z tej klasy mamy bezpośredni dostęp do wszystkich pól nagłówkowych HTTP. Ogólnym sposobem na to jest para metod `String getHeaderFieldKey(int n)` i `String getHeaderField(int n)` zwracających nazwę i wartość `n`-tego pola. Jeżeli nie ma pola o takim numerze, to metody te zwracają `null`. Jest też w tej klasie cały zestaw metod specyficz-

nych dla konkretnych pól nagłówkowych i informacji z pierwszego wiersza odpowiedzi oraz odpowiednich metod ustalających te wartości w wysyłanym żądaniu np. (spora część tych metod odziedziczona jest po `URLConnection`):

— `int getResponseCode()` — kod odpowiedzi HTTP,
— `String getResponseMessage()` — tekst powiązany z kodem odpowiedzi,
— `int getContentLength` — pole `Content-Length`,
— `long getDate` — pole `Date`
— `publicString getContentType()` — pole `Content-Type`
— `void setRequestMethod(String method)` — ustawia użytą w żądaniu metodę HTTP (GET, POST, HEAD).

Zawartość zasobu (część zasadniczą odpowiedzi) możemy pobrać w taki sam sposób jak za pomocą klasy `URL` — metoda `getInputStream` jest metodą, którą wywołuje metoda`openStream` w przypadku URL-a HTTP.

Domyślnie klasa `HttpURLConnection` podąża automatycznie za przekierowaniami nie zwracając o tym żadnej informacji. Możemy to zachowanie kontrolować na dwa sposoby:

— `static void setFollowRedirects(boolean set)` — ustala zachowanie dla całej klasy,
— `void setInstanceFollowRedirects(boolean followRedirects)` — ustala zachowanie dla danego obiektu.

Przykładem użycia tej klasy jest kod na listingu B.10. Powoduje on wysłania żądania HEAD, a następnie zwrócenie informacji o typie odpowiedzi. Jeżeli kod jest inny niż 200, to kod ten jest wyświetlany razem ze skojarzonym z nim komunikatem. Jeżeli było przekierowanie, to wyświetlony jest także cel przekierowania.

Listing B.10. Użycie metody HEAD do wyświetlenia informacji o typie pliku

```
  void info(String addr)
2   URL url = new URL(addr);
    HttpURLConnection httpConn =
4           (HttpURLConnection) url.openConnection();
    httpConn.setInstanceFollowRedirects(false);
6   httpConn.setRequestMethod("HEAD");
    int respCode = httpConn.getResponseCode();
8   if (httpConn.getResponseCode() == 200) {
      System.out.println(httpConn.getContentType());
10  } else {
      System.out.println(httpConn.getResponseCode()
12            + "␣" + httpConn.getResponseMessage());
      if (respCode == 301 || respCode == 302
```

```
14              || respCode == 303) {
          System.out.println("Location: "
                  + httpConn.getHeaderField(
                  "Location"));
18      }
      }
20  }
```

Dodatek C

Programy narzędziowe i diagnostyczne

C.1. `ifconfig` 182
C.2. `netstat` 183
C.3. `ping` 183
C.4. `traceroute` 184
C.5. `host`, `nslookup`, `dig` 185
C.6. `tcpdump` 186
C.7. Wireshark 187
C.8. `tcpflow` 188
C.9. `telnet` 188
C.10. `netcat` 188
C.11. Informacje diagnostyczne w aplikacjach użytkowych . . 188
C.12. `strace -e trace=network` 188

Dodatek ten omawia kilka programów mogących przydać się podczas pisania programów sieciowych.

C.1. `ifconfig`

Program `ifconfig` pozwala na konfigurację interfejsów sieciowych oraz wyświetlanie informacji o nich w systemach typu Unix. Pozwala poznać m.in. adres IP i maskę sieciową związane z tym interfejsem. Wywołanie `ifconfig` bez parametrów wyświetla informacje o aktywnych interfejsach sieciowych. Wywołanie z nazwą pojedynczego interfejsu wyświetla informacje o tym właśnie interfejsie. W pozostałych przypadkach `ifconfig` służy do konfigurowania interfejsów. Poniższy przykład wyświetla informacje o trzech interfejsach sieciowych: działającym `eth0`, niedziałającym eth1 i pętli zwrotnej lo

```
$ /sbin/ifconfig -a
eth0      Link encap:Ethernet  HWaddr 00:15:17:60:0c:2c
          inet addr:212.182.0.171  Bcast:212.182.1.255  Mask:255.255.254.0
          inet6 addr: fe80::215:17ff:fe60:c2c/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:273947992 errors:0 dropped:43896 overruns:0 frame:0
          TX packets:164981545 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:100
          RX bytes:579529186 (552.6 MiB)  TX bytes:3176304929 (2.9 GiB)
          Memory:b8820000-b8840000

eth1      Link encap:Ethernet  HWaddr 00:15:17:60:0c:2d
          BROADCAST MULTICAST  MTU:1500  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)
          Memory:b8800000-b8820000

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:16436  Metric:1
          RX packets:909544 errors:0 dropped:0 overruns:0 frame:0
          TX packets:909544 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:549119024 (523.6 MiB)  TX bytes:549119024 (523.6 MiB)
```

Odpowiednikiem `ifconfig` w systemie Windows jest `ipconfig`.

C.2. netstat

Program dostępny zarówno w systemach uniksowych, jak i w systemach z rodziny Windows. Ma kilka opcji umożliwiających wyświetlenie podobnych informacji do innych popularnych programów, np.:

— `-i` — wyświetla informacje o interfejsach sieciowych,
— `-r` — informacja o tablicach trasowania.

Szczególnie przydatny jest do diagnozowania stanów połączeń (patrz TCP — podrozdział 2.3.1.2).

Przydatnymi opcjami są:

— `-a` — wyświetla także gniazda nasłuchujące,
— `-p` — wyświetla pid i nazwę programu, do którego należy gniazdo,
— `--protocol=`*`family`* — wyświetla gniazda tylko dla danej rodziny protokołów, np, `--protocol=ip`, czy `--protocol=unix`.

Przykładowe wywołanie z wybranymi wierszami:

```
$ netstat --protocol=ip -a
Active Internet connections (servers and established)
Proto Recv-Q Send-Q Local Address   Foreign Address   State
tcp   0    0 *:imaps             *:*                LISTEN
tcp   0    0 *:pop3s             *:*                LISTEN
tcp   0    0 *:mysql             *:*                LISTEN
tcp   0    0 matrix.umcs.lub:www baiduspider-18:1263 SYN_RECV
tcp   0    0 *:smtp              *:*                LISTEN
tcp   0    0 localhost:1019      *:*                LISTEN
tcp   0    0 *:nqs               *:*                LISTEN
tcp   0    0 matrix.umcs.l:34610 host-80-252-0-1:www TIME_WAIT
tcp   0  132 matrix.umcs.lub:938 89-75-109-228.:1223 ESTABLISHED
tcp   0    0 matrix.umcs.lub:938 89-75-109-228.:1308 ESTABLISHED
udp   0    0 *:5678              *:*
udp   0    0 *:883               *:*
udp   0    0 matrix.umcs.lub:ntp *:*
udp   0    0 localhost:ntp       *:*
```

Na listingu widać kilka programów nasłuchujących na połączenie TCP (stan `LISTEN`), jeden program tuż przed zaakceptowaniem takiego połączenia (stan `SYN_REC`), jeden program, który niedawno aktywnie zamknął połączenie TCP (`TIME_WAIT`), dwa programy w trakcie nawiązanego połączenia TCP i kilka programów oczekujących na dowiązanym porcie UDP.

C.3. ping

Program `ping` pozwala sprawdzić połączenie sieciowe z zadanym hostem. Program wysyła komunikaty ICMP *echo request* i czeka na odpowiedź —

komunikaty ICMP *echo reply*. Wyświetla informacje o liczbie zgubionych pakietów i czasie potrzebnym na odpowiedź.

Przykład wywołania `ping`:

```
$ ping www.google.pl
PING www-cctld.l.google.com (209.85.173.94) 56(84) bytes of data.
64 bytes from lpp01m01-in-f94.1e100.net (209.85.173.94): icmp_seq=1 ttl=48
      time=57.4 ms
64 bytes from lpp01m01-in-f94.1e100.net (209.85.173.94): icmp_seq=2 ttl=48
      time=57.5 ms
64 bytes from lpp01m01-in-f94.1e100.net (209.85.173.94): icmp_seq=3 ttl=48
      time=57.6 ms
64 bytes from lpp01m01-in-f94.1e100.net (209.85.173.94): icmp_seq=4 ttl=48
      time=57.6 ms
64 bytes from lpp01m01-in-f94.1e100.net (209.85.173.94): icmp_seq=5 ttl=48
      time=57.5 ms
64 bytes from lpp01m01-in-f94.1e100.net (209.85.173.94): icmp_seq=6 ttl=48
      time=57.4 ms
^C
--- www-cctld.l.google.com ping statistics ---
6 packets transmitted, 6 received, 0% packet loss, time 5024ms
rtt min/avg/max/mdev = 57.476/57.551/57.686/0.318 ms
```

Dokument [8] wymaga, aby każdy host implementował obsługę komunikatów *echo request* i *echo reply*, ale ze względów bezpieczeństwa nie zawsze tak jest.

C.4. `traceroute`

Program `traceroute` dostępny w systemach uniksowych podobnie jak `ping` sprawdza połączenie sieciowe z wybranym hostem. Dodatkowo stara się też wyznaczyć dokładną trasę. Działanie oparte jest na wysyłaniu do hosta docelowego pakietów UDP z pewnym dużym numerem portu (tak żeby prawdopodobieństwo działania jakiejś usługi na tym porcie było małe) oraz polem TTL pakietu IP ustawianym na kolejne wartości (1, 2, ...). Trasa wyznaczana jest na podstawie komunikatów ICMP *time exceeded* przychodzących od kolejnych hostów.

Brak odpowiedzi na wysłany pakiet zaznaczany jest znakiem `*`.

Przykładowe wywołanie:

```
$ traceroute google.pl
traceroute to google.pl (209.85.173.94), 30 hops max, 60 byte packets
 1  vlan.109.ex4200-mfi-s124-1.lubman.net.pl (212.182.0.1)  5.560 ms  ...
 2  port-channel6.loki.lubman.net.pl (87.246.210.64)  0.482 ms  0.521 ...
 3  gi3-6.vidar.lubman.net.pl (87.246.210.71)  0.461 ms  0.519 ms  0.5...
 4  gi3-6.renfri.lubman.net.pl (212.182.56.196)  1.033 ms  1.318 ms  1...
 5  ae0x1035.herger.lubman.net.pl (212.182.63.50)  0.498 ms  0.554 ms ...
 6  z-lublina.poznan-gw1.10Gb.rtr.pionier.gov.pl (212.191.224.81)  7.4...
```

```
 7  pionier.rt1.poz.pl.geant.net (62.40.124.181)  7.716 ms  7.671 ms  ...
 8  xe-2-1-1.rt1.fra.de.geant.net (62.40.112.61)  36.664 ms  36.726 ms...
 9  google-gw.rt1.fra.de.geant.net (62.40.125.202)  36.715 ms  36.632 ...
10  209.85.241.110 (209.85.241.110)  36.946 ms  36.950 ms 209.85.240.6...
11  72.14.239.60 (72.14.239.60)  37.211 ms  37.151 ms 72.14.239.62 (72...
12  209.85.242.187 (209.85.242.187)  45.798 ms  45.778 ms  45.788 ms
13  209.85.240.88 (209.85.240.88)  85.359 ms  51.977 ms  51.928 ms
14  72.14.236.228 (72.14.236.228)  57.600 ms  57.712 ms  57.719 ms
15  72.14.233.174 (72.14.233.174)  57.120 ms  57.424 ms 72.14.233.170 ...
16  * 216.239.49.245 (216.239.49.245)  57.441 ms *
17  lpp01m01-in-f94.1e100.net (209.85.173.94)  57.498 ms  57.355 ms   ...
```

Odpowiednikiem `traceroute` w systemie Windows jest `tracert`.

C.5. host, nslookup, dig

Programami umożliwiającymi wysyłanie zapytań DNS są `host`, `nslookup`, `dig`. Poniżej przedstawione są przykłady wywołania polecenia z różnymi użytymi typami rekordów zasobów. W odpowiedziach pokazane są tylko wybrane wiersze.

Przykład zapytania serwer pocztowy dla domeny:

```
$ dig umcs.pl MX

;; ANSWER SECTION:
umcs.pl.                14400   IN      MX      5 mail.umcs.lublin.pl.

;; AUTHORITY SECTION:
umcs.pl.                14400   IN      NS      ns2.lublin.pl.
umcs.pl.                14400   IN      NS      dns1.umcs.lublin.pl.
umcs.pl.                14400   IN      NS      ns1.lublin.pl.

;; ADDITIONAL SECTION:
mail.umcs.lublin.pl.    14076   IN      A       212.182.74.26
ns1.lublin.pl.          15283   IN      A       212.182.63.66
ns2.lublin.pl.          15283   IN      A       212.182.63.70
dns1.umcs.lublin.pl.    4755    IN      A       212.182.74.2
```

Sekcja `ANSWER` zawiera właściwą odpowiedź. Sekcja `AUTHORITY` informację o serwerach autorytatywnych dla danej domeny, a sekcja `ADDITIONAL` — dodatkowe informacje, które mogą być przydatne (tutaj: adresy IP dla nazw domenowych z poprzednich sekcji).

Przykład zapytania o adres IP dla zadanej nazwy domenowej:

```
$ dig www.iana.org A

;; QUESTION SECTION:
;www.iana.org.                  IN      A
```

```
;; ANSWER SECTION:
www.iana.org.           600     IN      CNAME   ianawww.vip.icann.org.
ianawww.vip.icann.org.  30      IN      A       192.0.32.8

;; AUTHORITY SECTION:
vip.icann.org.          3600    IN      NS      gtm1.lax.icann.org.
vip.icann.org.          3600    IN      NS      gtm1.mdr.icann.org.
vip.icann.org.          3600    IN      NS      gtm1.dc.icann.org.
```

Przykład zapytania o adres domenowy dla zadanego adresu IP:

```
;; ANSWER SECTION:
171.0.182.212.in-addr.arpa. 22444 IN    PTR     matrix.umcs.lublin.pl.

;; AUTHORITY SECTION:
0.182.212.in-addr.arpa. 22444   IN      NS      ns1.lublin.pl.
0.182.212.in-addr.arpa. 22444   IN      NS      ns2.lublin.pl.
0.182.212.in-addr.arpa. 22444   IN      NS      ns.icm.edu.pl.

;; ADDITIONAL SECTION:
ns.icm.edu.pl.          5330    IN      A       212.87.14.39
ns1.lublin.pl.          14932   IN      A       212.182.63.66
ns2.lublin.pl.          14932   IN      A       212.182.63.70
```

C.6. `tcpdump`

Program `tcpdump` pozawala nam śledzić informacje z nagłówków przesyłanych pakietów. Warunki zadane argumentami wiersza poleceń pozwalają nam ograniczyć się do wybranych pakietów (np. na podstawie interfejsu, hosta, protokołu lub portu).

Przykładowa sesja:

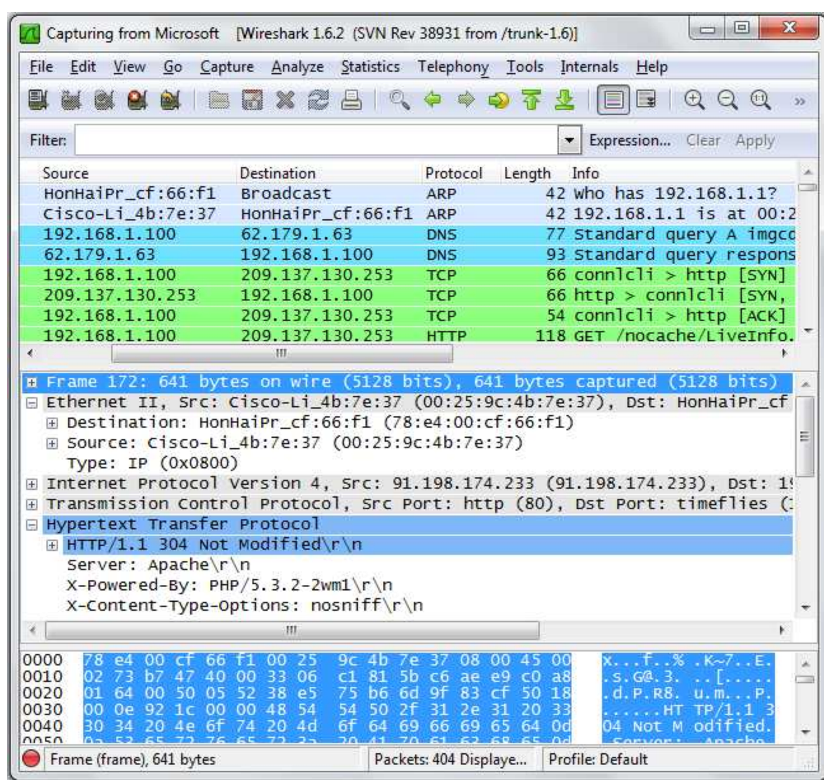
```
$ tcpdump "host matrix.umcs.lublin.pl and (udp or icmp)"
listening on eth0, link-type EN10MB (Ethernet), capture size 96 bytes
19:40:50.387499 IP praca.local.59053 > matrix.umcs.lublin.pl.7895:
  UDP, length 10000
19:40:50.387513 IP praca.local > matrix.umcs.lublin.pl: udp
19:40:50.387517 IP praca.local > matrix.umcs.lublin.pl: udp
19:40:50.387521 IP praca.local > matrix.umcs.lublin.pl: udp
19:40:50.387524 IP praca.local > matrix.umcs.lublin.pl: udp
19:40:50.387536 IP praca.local > matrix.umcs.lublin.pl: udp
19:40:50.387541 IP praca.local > matrix.umcs.lublin.pl: udp
19:40:50.388922 IP praca.umcs.lublin.pl > praca.local:
  ICMP matrix.umcs.lublin.pl udp port 7895 unreachable, length 556
```

Na powyższej sesji widać przesłanie jednego datagramu UDP i odesłanie jednego komunikatu ICMP. Datagram UDP przesłany jest jako kilka fragmentów IP.

C.7. Wireshark

Innym popularnym umożliwiającym analizę przesyłanych pakietów jest wieloplatformowy Wireshark. Różni się od dotychczas omówionych programów tym, że jest programem z graficznym interfejsem użytkownika. Rozpoznaje ogromną liczbę protokołów różnych warstw (od warstwy łącza danych wzwyż) i formatów sieciowych[1]. Oprócz funkcji związanych z przechwytywaniem pakietów (np. filtry przechwytywanych pakietów) ma też sporo ułatwiających pracę opcji związanych z interfejsem użytkownika (np. filtry wyświetlania pakietów).

Przykładowe okno programu może wyglądać tak jak na rysunku C.1.



Rysunek C.1. Wireshark

[1] Są wymienione na stronie `http://www.wireshark.org/docs/dfref/`.

C.8. `tcpflow`

Programem umożliwiającym przechwytywanie danych przesyłanych za pomocą protokołu TCP (bez informacji z nagłówków segmentów, czy informacji o podziale na segmenty) jest `tcpflow`. Dla każdego połączenia program tworzy dwa pliki tekstowe — do zapisania komunikacji w obie strony. Nazwy plików są złożone z adresów i portów źródłowych i docelowych.

C.9. `telnet`

Programem umożliwiającym symulowanie klienta tekstowego protokołu warstwy aplikacji korzystającego z TCP jest `telnet`. Do wierszy danych pobranych od użytkownika dokleja on sekwencję CRLF i przesyła do serwera. Wszystkie odpowiedzi serwera są wyświetlane bezpośrednio na standardowe wyjście.

Wszystkie sesje z protokołami pocztowymi i protokołem HTTP przedstawione w tej książce mogłyby być symulowane programem `telnet`.

C.10. `netcat`

Programem mającym wszystkie możliwości programu telnet, ale rozszerzającym go o możliwość symulowania serwera i obsługę UDP, jest `netcat` uruchamiany często poleceniem `nc`.

C.11. Informacje diagnostyczne w aplikacjach użytkowych

Niektóre aplikacje korzystające z protokołów warstwy aplikacji same udostępniają różne możliwości diagnostyczne. Na przykład większość programów pocztowych umożliwia obejrzenie kompletnej wiadomości pocztowej jako zwykłego tekstu ze wszystkimi nagłówkami.

Innym przykładem jest wtyczka HTTP Live Headers do przeglądarki internetowej Firefox. Wtyczka ta umożliwia oglądanie nagłówków wszystkich przesyłanych przez przeglądarkę żądań i otrzymywanych odpowiedzi.

C.12. `strace -e trace=network`

Do testowania programów sieciowych mogą się też przydać programy mające ogólniejsze przeznaczenie, ale z opcjami dotyczącymi sieci. Przykładem może być `strace` wyświetlający wywołania systemowe zadanego programu

i opcja `-e trace=network` wyświetlająca wywołania systemowe związane z siecią.

Na przykład, jeżeli chcemy wiedzieć jak działa `ping`, to możemy się tego dowiedzieć tak:

```
$ strace -e trace=network ping -c 1 87.246.208.8
socket(PF_INET, SOCK_RAW, IPPROTO_ICMP) = 3
socket(PF_INET, SOCK_DGRAM, IPPROTO_IP) = 4
connect(4, {sa_family=AF_INET, sin_port=htons(1025), sin_addr=inet_addr(...
getsockname(4, {sa_family=AF_INET, sin_port=htons(49503), sin_addr=inet_...
setsockopt(3, SOL_RAW, ICMP_FILTER, ~(ICMP_ECHOREPLY|ICMP_DEST_UNREACH|I...
setsockopt(3, SOL_IP, IP_RECVERR, [1], 4) = 0
setsockopt(3, SOL_SOCKET, SO_SNDBUF, [324], 4) = 0
setsockopt(3, SOL_SOCKET, SO_RCVBUF, [65536], 4) = 0
getsockopt(3, SOL_SOCKET, SO_RCVBUF, [17180000256], [4]) = 0
setsockopt(3, SOL_SOCKET, SO_TIMESTAMP, [1], 4) = 0
setsockopt(3, SOL_SOCKET, SO_SNDTIMEO, "\1\0\0\0\0\0\0\0\0\0\0\0\0\0\0...
setsockopt(3, SOL_SOCKET, SO_RCVTIMEO, "\1\0\0\0\0\0\0\0\0\0\0\0\0\0\0...
sendmsg(3, {msg_name(16)={sa_family=AF_INET, sin_port=htons(0), sin_addr...
recvmsg(3, {msg_name(16)={sa_family=AF_INET, sin_port=htons(0), sin_addr...
```

Bibliografia

[1] H. Alvestrand: *IETF policy on character sets and languages.* RFC 2277, Internet Engineering Task Force, January 1998.

[2] A. Barth: *HTTP State Management Mechanism.* RFC 6265, Internet Engineering Task Force, April 2011.

[3] T. Berners-Lee: *The original http as defined in 1991.* `http://www.w3.org/Protocols/HTTP/AsImplemented.html`

[4] T. Berners-Lee, R. Fielding, H. Frystyk: *Hypertext transfer protocol — HTTP/1.0.* RFC 1945, Internet Engineering Task Force, May 1996.

[5] T. Berners-Lee, R. Fielding, L. Masinter: *Uniform resource identifier (URI): generic syntax.* RFC 3986, Internet Engineering Task Force, January 2005.

[6] T. Berners-Lee, L. Masinter, M. McCahill: *Uniform resource locators (url).* RFC 1738, Internet Engineering Task Force, December 1994.

[7] R. Braden: *Requirements for internet hosts — application and support.* RFC 1123, Internet Engineering Task Force, October 1989.

[8] R. Braden: *Requirements for internet hosts — communication layers.* RFC 1122, Internet Engineering Task Force, October 1989.

[9] J. Bylina, B. Bylina: *Przegląd języków i paradygmatów programowania.* UMCS, Lublin 2011.

[10] M. R. Crispin: *INTERNET MESSAGE ACCESS PROTOCOL — VERSION 4rev1.* RFC 3501, Internet Engineering Task Force, March 2003.

[11] D. H. Crocker: *Internet Mail Architecture.* RFC 5321, Internet Engineering Task Force, July 2009.

[12] D. H. Crocker: *Standard for the format of ARPA internet text messages.* RFC 822, Internet Engineering Task Force, August 1982.

[13] S. E. Deering, R. Hinden: *Internet protocol, version 6 (IPv6) specification.* RFC 2460, Internet Engineering Task Force, December 1998.

[14] A. B. Downey: *Python for Software Design: How to Think Like a Computer Scientist.* Cambridge University Press 2009. `http://greenteapress.com/thinkpython/thinkpython.html`

[15] D. Eastlake 3rd, A. Panitz: *Reserved top level DNS names.* RFC 2606, Internet Engineering Task Force, June 1999.

[16] R. Fielding, J. Gettys, J. C. Mogul, H. Frystyk, L. Masinter, P. J. Leach, T. Berners-Lee: *Hypertext transfer protocol — HTTP/1.1.* RFC 2616, Internet Engineering Task Force, June 1999.

[17] J. Franks, P. Hallam-Baker, J. Hostetler, S. Lawrence, P. J. Leach, A. Luotonen, L. Stewart: *HTTP authentication: Basic and digest access authentication.* RFC 2617, Internet Engineering Task Force, June 1999.

[18] N. Freed, N. Borenstein: *Multipurpose internet mail extensions (MIME) part one: Format of internet message bodies.* RFC 2045, Internet Engineering Task Force, November 1996.

[19] N. Freed, N. Borenstein: *Multipurpose internet mail extensions (MIME) part two: Media types.* RFC 2046, Internet Engineering Task Force, November 1996.

[20] W. W. Gay: *Linux. Gniazda w programowaniu — w przykładach.* MIKOM, Warszawa 2001.

[21] R. Gilligan, Sue Thomson, J. Bound, J. McCann, W. R. Stevens: *Basic socket interface extensions for IPv6.* RFC 3493, Internet Engineering Task Force, March 2003.

[22] T. Hain: *Architectural implications of NAT.* RFC 2993, Internet Engineering Task Force, November 2000.

[23] E. R. Harold: *Java: programowanie sieciowe.* Wydawnictwo RM, 2001.

[24] I. Hickson: *Sending xhtml as text/html considered harmful.* `http://hixie.ch/advocacy/xhtml`

[25] R. Hinden, S. Deering: *Classless inter-domain routing (cidr): The internet address assignment and aggregation plan.* RFC 4632, Internet Engineering Task Force, August 2006.

[26] R. Hinden, S. Deering: *IP version 6 addressing architecture.* RFC 4291, Internet Engineering Task Force, February 2006.

[27] J. E. Hopcroft, B. Motwani, J. D. Ullman: *Wprowadzenie do teorii automatów, języków i obliczeń.* Wydawnictwo Naukowe PWN, 2005.

[28] M. T. Jones: *BSD Sockets Programming from a Multi-Language Perspective.* Charles River Media 2004.

[29] J. Klensin: *Simple Mail Transfer Protocol.* RFC 5321, Internet Engineering Task Force, October 2008.

[30] G. Klyne, C. Newman: *Date and time on the internet: Timestamps.* RFC 3339, Internet Engineering Task Force, July 2002.

[31] B. Krishnamurthy, J. C. Mogul, D. M. Kristol: *Key differences between HTTP/1.0 and HTTP/1.1.* `http://www8.org/w8-papers/5c-protocols/key/key.htmll`

[32] K. Kuczyński, R. Stęgierski: *Routing w sieciach IP.* UMCS, Lublin 2011.

[33] J. F. Kurose, K. W. Ross: *Computer Networking: A Top-Down Approach.* Addison-Wesley Publishing Company, USA, 5th edition, 2009.

[34] M. Lutz, D. Ascher: *Python. Wprowadzenie.* Helion, Gliwice 2002.

[35] A. N. Marine, J. F. Reynolds, G. Malkin: *FYI on questions and answers — answers to commonly asked "new internet user" questions.* RFC 1594, Internet Engineering Task Force, March 1994.

[36] A. Martelli, A. Martelli Ravenscroft, D. Ascher: *Python. Receptury.* Helion, Gliwice 2006.

[37] P. V. Mockapetris: *Domain names — concepts and facilities.* RFC 1034, Internet Engineering Task Force, November 1987.

[38] P. V. Mockapetris: *Domain names — implementation and specification.* RFC 1035, Internet Engineering Task Force, November 1987.

[39] K. Moore: *MIME (multipurpose internet mail extensions) part three: Message header extensions for Non-ASCII text.* RFC 2047, Internet Engineering Task Force, November 1996.

[40] J. Myers, M. P. Rose: *Post office protocol — version 3.* RFC 1939, Internet Engineering Task Force, May 1996.
[41] P. Norton i inni: *Python. Od podstaw.* Helion, Gliwice 2006.
[42] J. Palme: *Common Internet Message Headers.* RFC 2076, Internet Engineering Task Force, February 1997.
[43] M. Pilgrim: *Zanurkuj w Pythonie.* `http://pl.wikibooks.org/wiki/Zanurkuj_w_Pythonie`
[44] E. Pitt: *Fundamental networking in Java.* Springer, 2006.
[45] J. B. Postel: *Internet control message protocol.* RFC 792, Internet Engineering Task Force, September 1981.
[46] J. B. Postel: *Internet protocol.* RFC 791, Internet Engineering Task Force, September 1981.
[47] J. B. Postel: *Simple mail transfer protocol.* RFC 821, Internet Engineering Task Force, August 1982.
[48] J. B. Postel. *Transmission control protocol.* RFC 793, Internet Engineering Task Force, September 1981.
[49] J. B. Postel: *User datagram protocol.* RFC 768, Internet Engineering Task Force, August 1980.
[50] P. Resnick: *Internet Message Format.* RFC 2822, Internet Engineering Task Force, April 2001.
[51] K. G. Ramakrishnan, S. Floyd, D. L. Black: *The addition of explicit congestion notification (ECN) to IP.* RFC 3168, Internet Engineering Task Force, September 2001.
[52] Y. Rekhter, B. Moskowitz, D. Karrenberg, G. J. de Groot, E. Lear: *Address allocation for private internets.* RFC 1918, Internet Engineering Task Force, February 1996.
[53] P. Resnick: *Internet Message Format.* RFC 5322, Internet Engineering Task Force, October 2008.
[54] P. Srisuresh, K. Egevang: *Traditional IP network address translator (traditional NAT).* RFC 3022, Internet Engineering Task Force, January 2001.
[55] P. Srisuresh, M. Holdrege: *IP network address translator (NAT) terminology and considerations.* RFC 2663, Internet Engineering Task Force, August 1999.
[56] W. R. Stevens: *UNIX — programowanie usług sieciowych.* Wydawnictwa Naukowo-Techniczne 1999.
[57] J. Swift: *Podróże Guliwera* (ang. *Gulliver's Travels*). Londyn 1726.
[58] A. S. Tanenbaum: *Sieci komputerowe.* Helion, Gliwice, 2004.
[59] R. Troost, S. Dorner, K. Moore, W. Pedrycz: *Communicating presentation information in internet messages: The Content-Disposition header field.* RFC 2183, Internet Engineering Task Force, August 1997.
[60] K. Wall: *Linux. Programowanie — w przykładach.* MIKOM, Warszawa 2000.
[61] C. Wong: *HTTP. Leksykon kieszonkowy.* O'Reilly, 2001.
[62] D. Wood: *Programming Internet email.* Animal Series. O'Reilly, 1999.
[63] `http://curl.haxx.se/`
[64] `http://docs.python.org/tutorial/`
[65] `http://python.org/doc/`

Skorowidz

A, 46
AAAA, 46
accept, 79
ACK, 32, 34
addrinfo, 48
adres, 44
  domenowy, 40, 46–48
  e-mail, 111
  IP, 22, 24–30, 32, 40, 44–48
    prywatny, 26, 27
  IPv6, 28
  uogólniony, 57
AF_INET, 45
AF_INET6, 45
algorytm Nagle'a, 34
algorytm opóźnionego potwierdzenia, 34
API, 20
argc, 3
argv, 3

big-endian, 40
bind, 30, 43, 56

C (język programowania), 2, 40, 43
chmod, 10
CIDR, 26
close, 10, 35, 80
connect, 29, 33, 43, 58, 70
czas życia, 24, 28

dane pozapasmowe, 32
datagram, 19–21, 54
  IP, 22, 27, 28
demon, 105
demonizacja procesu, 105
deskryptor pliku, 5, 9, 11
DF, 24
dig, 47
DNS, 39, 40, 46, 47
dobrze znany port, 27, 30, 41
domena najwyższego poziomu, 46
dowiązanie twarde, 12

ECONNREFUSED, 29
EHOSTUNREACH, 29
environ, 4
errno, 29
errno, 8
exec (rodzina funkcji), 98
exit, 5, 97
_exit, 5, 97

fchmod, 10
fcntl, 90
FD_CLR, 102
FD_ISSET, 102
FD_SET, 102
FD_ZERO, 102
FIN, 32, 34
fork, 96, 103
FQDN, 46
fragment, 19, 24
fragmentacja, 19, 24
freeaddrinfo, 49
fstat, 10
funkcje systemowe, 6, 8

gcc, 2
getaddrinfo, 48, 49
getenv, 4
gethostbyaddr, 47
gethostbyaddr_r, 48
gethostbyname, 47
gethostbyname2, 48
gethostbyname2_r, 48
gethostbyname_r, 48
getnameinfo, 48

getpeername, 84
getpid, 97
getppid, 97
getservbyname, 42
getservbyport, 42
getsockname, 84
getsockopt, 22, 88
gniazdo, 20, 28, 30
  datagramowe, 54
  nasłuchujące, 79
  strumieniowe, 68
  surowe, 23, 29
  TCP, 29
  UDP, 29
  w dziedzinie Unix, 43
GNU libc, 2

h_errno, 47
host, 47
hostent, 48
htonl, 40
htons, 40, 45
HTTP, 125

IANA, 41
identyfikator hosta, 25
identyfikator sieci, 25
ifconfig, 26
IMAP, 120
INADDR_BROADCAST, 45
INADDR_LOOPBACK, 28
INADDR_NONE, 45
inet_addr, 45
INET_ADDRSTRLEN, 45
inet_aton, 45
inet_ntoa, 45
inet_ntop, 45
inet_pton, 45
info, 8
interfejs sieciowy, 25
ioctl, 90
IP_DONTFRAG, 24
IP_DONTFRAGMENT, 24
IP_HDRINCL, 23
IP_OPTIONS, 24
IP_TOS, 23
IP_TTL, 24
ipconfig, 26
IPPROTO_ICMP, 29
IPPROTO_IP, 88
IPPROTO_TCP, 88

Java, 166

kill, 99
klasa adresów IP, 26
klasa adresu IP, 25, 26
klient, 29, 30
klient-serwer, 29
kod zakończenia, *zob.* status zakończenia
komunikat ICMP, 29

listen, 33, 79
little-endian, 40
localhost, 46
lseek, 10

main, 3
man, 6
maska podsieci, 26
MF, 24
MIME, 113
mkdir, 10
model odniesienia
  OSI, 19, 20
  TCP/IP, 19, 20
MSG_DONTROUTE, 94
MSG_DONTWAIT, 94
MSG_EOR, 94
MSG_OOB, 32, 94
MSG_PEEK, 94
MSG_WAITALL, 94
MTU, 24
multipleksacja wejścia/wyjścia, 100
MX, 47

nagłówek
  datagramu IP, 22, 23
  HTTP, 127
  TCP, 30–32
  UDP, 30
  wiadomości pocztowej, 112
NAPT, 27
NAT, 26, 27
nawiązywanie połączenia, 33

nazwa domenowa, 46
netstat, 32
nslookup, 47
ntohl, 40
ntohs, 40

O_ASYNC, 90
O_NONBLOCK, 11, 90
ogólnie znany port, *zob.* dobrze znany port
okno przesuwne, 30, 32
opcje gniazd, 22, 23
opcje IP, 24
open, 10

półzamknięcie połączenia, 84
pętla zwrotna, 27, 46
pakiet, 19
pakiet IP, *zob.* datagram IP
pełna nazwa domenowa, 46
perror, 9
ping, 29
plik
  /etc/hosts, 47
  /etc/nsswitch.conf, 47
  /etc/resolv.conf, 47
  /etc/services, 41
pliki blokujące, 11
poczta elektroniczna, 110
podsieć, 26
POP3, 118
port, 26, 27, 29–31, 41–43
  dobrze znany, *zob.* dobrze znany port
  ogólnie znany, *zob.* dobrze znany port
  zarejestrowany, 41
POSIX, 2
prawa dostępu, *zob.* uprawnienia
pre-fork, 103
proces, 96
protokół, 18, 21, 29
  bezpołączeniowy, 22
  FTP, 27
  ICMP, 21, 24, 28, 29
  IP, 20–22, 28, 29
  IPv4, 22, 28, 43
  IPv6, 22, 24, 28, 43, 48
  niezawodny, 30
  połączeniowy, 30
  strumieniowy, 31
  TCP, 22, 24, 29–31, 41, 68
  transportowy, 30
  UDP, 22, 24, 29, 35, 54
  zawodny, 22
PSH, 32
PTR, 47
Python, 146

ramka, 19
read, 10, 34, 70
recv, 70, 94
recvfrom, 63, 94
rekord zasobów, 46
rename, 10
resolver, 40, 47
rmdir, 10
RST, 29, 32

segment, 20
  TCP, 31
select, 63, 101, 104
send, 32, 70, 94
sendto, 32, 63, 94
servent, 42
serwer, 29, 30, 33, 54, 60, 76
  współbieżny, 30, 103
setsid, 105
setsockopt, 88
shutdown, 35, 83
sieciowa kolejność bajtów, 40, 44
sigaction, 100
signal, 100
sigprocmask, 100
SMTP, 116
SO_DEBUG, 91
SO_DONTROUTE, 91
SO_ERROR, 91
SO_KEEPALIVE, 91, 92
SO_LINGER, 91
SO_OOBINLINE, 91, 94
SO_RCVBUF, 91, 92
SO_RCVTIMEO, 93
SO_REUSEADDR, 92
SO_SNDBUF, 91, 92
SO_SNDTIMEO, 93

SOCK_RAW, 23
socket, 29, 30, 56
SOL_SOCKET, 88
standardowe wejście, 5
standardowe wyjście, 5
standardowe wyjście błędów, 6
standardowe wyjście diagnostyczne, *zob.* standardowe wyjście błędów
stat, 10
status zakończenia, 5
sterowanie przepływem, 30
suma kontrolna, 24, 32
sygnał, 99
SYN, 32, 34
systemowa kolejność bajtów, 40

TCP/IP, 21, 27, 40
TCP_NODELAY, 34
terminal sterujący, 105
TOS, 23
traceroute, 29
TTL, 24
typ usługi, 23

umask, 11, 107
unlink, 10
uprawnienia, 11
URG, 32
URL, 124

wait, 97
wait3, 98
wait4, 98
waitpid, 97
warstwa, 18, 19, 21, 27, 29
  aplikacji, 20, 27, 30, 110
  dostępu do sieci, 20
  fizyczna, 19
  host-sieć, *zob.* warstwa dostępu do sieci
  kanałowa, *zob.* warstwa łącza danych
  łącza danych, 19, 24
  prezentacji, 20
  sesji, 20
  sieciowa, 19–22, 27, 29, 40
  transportowa, 20–22, 26, 27, 29, 30
  zastosowań, *zob.* warstwa aplikacji, 22
Windows, 24, 26, 41
write, 10, 34, 70
wskaźnik pliku, 11, 12
WWW, 124

zamykanie połączenia, 34, 80, 83
zwielokrotnianie wejścia/wyjścia, *zob.* multipleksacja wejścia/wyjścia