Systemy rozproszone

Skrypt do ćwiczeń laboratoryjnych

Cezary Sobaniec
Marek Libuda

v1.1
07/09/2006

Uczelnia On-Line
2006

Spis treści

1 Zdalne wywołanie procedur Sun RPC **5**
1.1 Wprowadzenie . . . 5
1.2 Proste wywołanie zdalnej procedury . . . 6
1.3 Przykład aplikacji RPC . . . 8
1.3.1 Definicja usługi . . . 8
1.3.2 Generator kodu `rpcgen` . . . 9
1.4 XDR . . . 12
1.4.1 Wprowadzenie . . . 12
1.4.2 Potoki XDR . . . 13
1.4.3 Filtry XDR . . . 15
1.4.4 Filtry XDR tworzone przez `rpcgen` . . . 16
1.5 Asynchroniczne RPC . . . 19
1.5.1 Przebieg zdalnego wywołania . . . 19
1.5.2 Proste wywołanie asynchroniczne . . . 19
1.5.3 Modyfikacja pieńka serwera . . . 21
1.5.4 Procesy potomne . . . 23
1.6 Wywołanie zwrotne . . . 26
1.6.1 Wprowadzenie . . . 26
1.6.2 Przykład aplikacji . . . 26
1.6.3 Procedura `svc_run()` . . . 28
1.6.4 Rejestracja tymczasowej usługi . . . 30
1.7 Kontrola praw dostępu . . . 32
1.8 Wątki w standardzie POSIX . . . 32
1.8.1 Wprowadzenie . . . 32
1.8.2 Zarządzanie wątkami . . . 34
1.8.3 Obsługa sygnałów . . . 34
1.8.4 Synchronizacja wątków . . . 35

2 Java RMI **41**
2.1 Podstawy . . . 41
2.1.1 Wprowadzenie . . . 41
2.1.2 Procesy . . . 41
2.1.3 Etapy konstrukcji aplikacji . . . 41
2.1.4 Przykład prostej aplikacji . . . 42
2.1.5 Zadanie . . . 45

2.2 Java RMI — przekazywanie parametrów . . . 45
2.2.1 Wprowadzenie . . . 45
2.2.2 Przekazanie przez referencję . . . 46
2.2.3 Przekazanie przez wartość . . . 46
2.2.4 Bezpieczeństwo . . . 47
2.2.5 Zadanie . . . 48

3 CORBA 49

3.1 Wprowadzenie . . . 49
3.2 Etapy tworzenia aplikacji . . . 49
3.3 Przykład prostej aplikacji . . . 50
3.3.1 Interfejs obiektu . . . 50
3.3.2 Kompilacja interfejsu . . . 50
3.3.3 Kod implementacji obiektu . . . 50
3.3.4 Program serwera . . . 51
3.3.5 Program klienta . . . 52
3.3.6 Kompilacja i uruchomienie programów . . . 53

4 Network File System 55

4.1 Konfiguracja klienta . . . 55
4.2 Konfiguracja serwera . . . 57
4.2.1 Uruchomienie serwera . . . 58
4.2.2 Rekonfiguracja . . . 58
4.2.3 Grupy sieciowe . . . 59
4.3 Mapowanie użytkowników . . . 59
4.3.1 Mapowanie statyczne . . . 60
4.3.2 Mapowanie dynamiczne . . . 60
4.4 WebNFS . . . 61
4.5 Automonter . . . 61
4.5.1 Zwiększanie dostępności . . . 62
4.5.2 Dynamiczna konfiguracja . . . 62

5 Pakiet Samba 65

5.1 Wprowadzenie . . . 65
5.2 Klient protokołu SMB . . . 66
5.3 Konfiguracja serwera . . . 68
5.3.1 Zmienne specjalne . . . 68
5.3.2 Program administracyjny `swat` . . . 69
5.3.3 Polecenie `smbstatus` . . . 69
5.3.4 Samba a sprawa polska . . . 69
5.4 Kontroler domeny . . . 70
5.4.1 Konfiguracja serwera . . . 70
5.4.2 Uwierzytelnianie użytkowników SMB . . . 71
5.4.3 Uwierzytelnianie użytkowników uniksowych . . . 71
5.5 Pakiet Winbind . . . 72
5.5.1 Konfiguracja . . . 72
5.5.2 Katalogi domowe użytkowników uniksowych . . . 73
5.6 Opis parametrów konfiguracyjnych . . . 74

6 Network Information Service **81**
6.1 Wprowadzenie . . . 81
6.2 Konfiguracja serwera . . . 81
6.3 Konfiguracja klienta . . . 83
6.4 Zmiana hasła użytkownika . . . 85
6.5 Serwery pomocnicze . . . 85
6.6 Współpraca NIS i NFS . . . 86
6.7 Bezpieczeństwo . . . 86

7 LDAP **89**
7.1 Konfiguracja serwera . . . 89
7.2 Edycja rekordów . . . 90
7.2.1 Podstawowe narzędzia . . . 90
7.2.2 Dostęp poprzez przeglądarkę . . . 93
7.2.3 Graficzna przeglądarka GQ . . . 93
7.2.4 Inne narzędzia . . . 93
7.3 Prawa dostępu . . . 94
7.4 Integracja z systemem . . . 94
7.4.1 Moduł `nss_ldap` . . . 94
7.4.2 Moduł `pam_ldap` . . . 95
7.5 Znaki narodowe . . . 95
7.6 Bezpieczeństwo — SSL/TLS . . . 96
7.7 Integracja z innymi aplikacjami . . . 96
7.7.1 Samba . . . 96

Skorowidz **99**

Bibliografia **101**

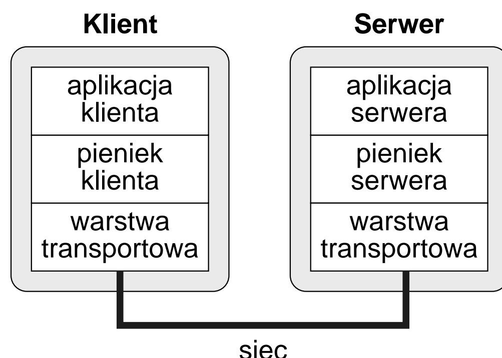
1 Zdalne wywołanie procedur Sun RPC

1.1 Wprowadzenie

Mechanizm zdalnego wywołania procedur (ang. *remote procedure call*) umożliwia wywoływanie procedur udostępnianych przez zdalne serwery w sposób zapewniający dużą przezroczystość wywołań w stosunku do wywołań procedur lokalnych. Koncepcja wywołań zdalnych została zrealizowana w wielu różnych środowiskach programowych. W ramach ćwiczeń zapoznamy się z konkretną realizacją: Sun RPC. Sun RPC jest zestawem bibliotek programowych i narzędzi wspomagających umożliwiających tworzenie aplikacji rozproszonych w języku C.

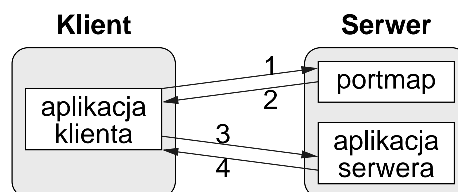
Rys. 1.1 pokazuje warstwy oprogramowania zaangażowane w realizację zdalnego wywołania procedury. Transparentność (przezroczystość) wywołania zdalnego zapewniają pieńki (ang. *stubs*) po stronie klienta i serwera. Pieniek klienta dostarcza lokalnie interfejsu zdalnej procedury. Po wywołaniu procedury w aplikacji klienta następuje przygotowanie komunikatu zawierającego kod wywołania procedury z argumentami i wysłanie go do serwera. Pieniek serwera odbiera komunikat, odpakowuje argumenty i wywołuje lokalną procedurę zaimplementowaną w aplikacji serwera. Wynik wywołania procedury jest następnie pakowany do nowego komunikatu i przesyłany do klienta. Pieniek klienta odbiera komunikat i przekazuje wyniki do aplikacji. Kod pieńków jest generowany automatycznie, co umożliwia programiście skoncentrowanie się na właściwej funkcjonalności aplikacji.

Procedury Sun RPC identyfikowane są po numerach. W typowych usługach sieciowych (np. WWW czy FTP), identyfikacja usługi następuje poprzez wskazanie dwóch wartości: adresu IP serwera i numeru portu (TCP lub UDP). W przypadku usługi RPC zdalna procedura identyfikowana jest czwórką wartości: adres IP serwera, numer usługi RPC, numer wersji usługi oraz numer procedury w ramach usługi. Oczywiście w systemie rozproszonym komunikacja ostatecznie musi sprowadzić się do zastosowania protokołów warstwy transportowej, co oznacza, że aplikacja klienta musi mieć możliwość pozyskania w jakiś sposób numerów portów do komunikacji z serwerem. Funkcjonalność tą zapewnia dodatkowa usługa *portmapper* (implementowana jako program o nazwie `portmap`), która pracuje na ogólnie znanym numerze portu (111). Zadaniem usługi jest odwzorowywanie numerów usług RPC na numery portów i nazwy protokołów transportowych poprzez które można się dostać do serwera zdalnej procedury. Wywołanie zdalnej procedury realizowane



Rys. 1.1: Warstwy oprogramowania realizujące zdalne wywołanie procedury RPC

jest więc zgodnie ze schematem przedstawionym na rys. 1.2. Aplikacja klienta (a dokładnie pieniek) wysyła zapytanie do usługi *portmapper* na port 111 pytając o usługę RPC o konkretnym numerze i wersji (1). Uzyskuje informacje o protokołach transportowych i numerach portów, poprzez które odpowiedni serwer RPC jest osiągalny (2). Następny komunikat zawiera już właściwe zlecenie wywołania zdalnej procedury i jest kierowany do właściwej aplikacji-serwera (3). Serwer RPC po wykonaniu zdalnej procedury odsyła wyniki (4). W kolejnych dowołaniach procedur można pominąć etap odpytywania usługi *portmapper*, zakładając, że konfiguracja serwera RPC nie zmieniła się.



Rys. 1.2: Schemat wywołania zdalnej procedury w Sun RPC

Szerszy opis RPC można znaleźć w [GD95]. Dokładna specyfikacja znajduje się w RFC 1050 [Mic88].

1.2 Proste wywołanie zdalnej procedury

Wywołanie zdalnej procedury może być realizowane na różnych poziomach szczegółowości. Najwyższy poziom zapewnia maksimum przezroczystości i faktycznie nie różni się od wywołania lokalnej procedury. Zejście niżej ujawnia pewne szczegóły realizacji zdalnego wywołania, dając jednakże możliwość zoptymalizowania jego przebiegu. Najniższy poziom pozwala na uzyskanie niestandardowych schematów realizacji zdalnego wywołania. Kosztem jest tu jednakże konieczność zapoznania się ze szczegółami implementacyjnymi pieńków oraz ich ręczna modyfikacja.

Na najwyższym poziomie wywołanie zdalnej procedury RPC można zrealizować z wykorzystaniem funkcji:

```
callrpc(char* hostname, u_long prognum,
        u_long versnum, u_long procnum,
        xdrproc_t inproc, char* in,
        xdrproc_t outproc, char* out)
```

Przykład 1.1: Klient usługi rusers

```
1  #include <stdio.h>
2  #include <rpc/rpc.h>
3  #include <rpcsvc/rusers.h>
4
5  int main(int argc, char **argv)
6  {
7    long nusers;
8    enum clnt_stat stat;
9
10   if (argc <= 1)
11   {
12     printf("Wywolanie:  %s <serwer>\n", argv[0]);
13     exit(1);
14   }
15
16   stat=callrpc(argv[1], RUSERSPROG, RUSERSVERS_3, RUSERSPROC_NUM,
17                xdr_void, NULL, xdr_u_long, &nusers);
18   if(stat != RPC_SUCCESS)
19   {
20     clnt_perrno(stat);
21     printf("\n");
22     exit(1);
23   }
24   printf("W systemie %s pracuje %d użytkowników.\n",
25          argv[1], nusers);
26 }
```

Przykładowy klient usługi `rusers` został przedstawiony w przykładzie 1.1.

Kompilację programu można przeprowadzić następującym wywołaniem:

```
# gcc -o rusers rusers.c
rusers.c: In function ‘main’:
rusers.c:19: warning: passing arg 5 of ‘callrpc’ from incompatible pointer type
rusers.c:19: warning: passing arg 7 of ‘callrpc’ from incompatible pointer type
rusers.c:19: warning: passing arg 8 of ‘callrpc’ from incompatible pointer type
```

Kompilacja kończy się ostrzeżeniami związanymi z niezgodnością typów danych niektórych argumentów. Dla uproszczenia zapisu przykładowego programu zrezygnowano bowiem z rzutowania argumentów do wymaganych przez funkcję `callrpc()` argumentów. Powstały program `rusers` można wykonać podając jako argument nazwę serwera:

```
# rusers localhost
RPC: Program not registered
```

Zgłoszony błąd wynika z nieobecności serwera usługi `rusers`. Każdy program RPC przed właściwym wywołaniem metody zdalnej musi bowiem dokonać odwzorowania numeru usługi RPC na numer portu w odpowiednim protokole transportowym. W powyższym przykładzie następuje odwołanie do usługi identyfikowanej jako `RUSERSPROG`. W pliku /usr/include/rpcsvc/rusers.h można znaleźć deklarację identyfikatora `RUSERSPROG` wskazującą na wartość 100002. Aktualnie zarejestrowane usługi można wyświetlić komendą `rpcinfo`:

```
# /usr/sbin/rpcinfo -p
   program vers proto   port
    100000    2   tcp    111  portmapper
    100000    2   udp    111  portmapper
    100007    2   udp    932  ypbind
    100007    1   udp    932  ypbind
    100007    2   tcp    935  ypbind
    100007    1   tcp    935  ypbind
```

W kolumnie `program` wyświetlany jest numer usługi RPC, kolumna `vers` opisuje wersję danej usługi, kolumna `proto` to nazwa protokołu transportowego, a kolumna `port` zawiera numer portu, na którym nasłuchuje odpowiedni serwer. W przykładowym systemie dostępne są więc dwie usługi: `portmapper` i `ypbind`. Serwer usługi `rusers` można uruchomić komendą:

```
# /usr/sbin/rpc.rusersd
```

której wykonanie powoduje uruchomienie w tle procesu o tej samej nazwie. Nowy serwer rejestruje swoją obecność w usłudze `portmapper`:

```
# /usr/sbin/rpcinfo -p
   program vers proto   port
    ...
    100002    3   udp  32769  rusersd
    100002    2   udp  32769  rusersd
```

Program `rusers` wyświetla teraz poprawnie liczbę procesów:

```
# rusers localhost
W systemie localhost pracuje 4 użytkowników.
```

1.3 Przykład aplikacji RPC

Poniższa kompletna aplikacja `rkill` ma za zadanie umożliwić zatrzymywanie zdalnych procesów poprzez wysłanie do nich sygnału KILL.

1.3.1 Definicja usługi

Definicja jest zapisana w pliku `rkill.x`:

```
program RKILL_PRG {
  version RKILL_VERSION_1 {
    int rkill(int pid) = 1;
  } = 1;
} = 0x21000001;
```

W przykładzie zdefiniowano usługę `RKILL_PRG` o numerze 0x21000001 (dziesiętnie: 553648129), która występuje w jednej wersji (identyfikowanej jako `RKILL_VERSION_1`). Usługa definiuje jedną funkcję o następującym nagłówku:

```
int rkill(int pid);
```

Wymagany jest więc jeden argument będący identyfikatorem zatrzymywanego procesu. Funkcja zwraca status zakończenia operacji wysłania sygnału.

1.3.2 Generator kodu rpcgen

Na podstawie przygotowanego pliku definicyjnego można automatycznie wygenerować pliki źródłowe przykładowej aplikacji:

```
# rpcgen -a -N rkill.x
# ls
Makefile.rkill  rkill.x         rkill_clnt.c    rkill_svc.c
rkill.h         rkill_client.c  rkill_server.c
```

Powstałe pliki reprezentują odpowiednio:

rkill.h	plik nagłówkowy z definicjami poszczególnych identyfikatorów,
rkill_client.c	szkielet aplikacji klienckiej,
rkill_server.c	szkielet serwera,
rkill_clnt.c	pieniek klienta,
rkill_svc.c	pieniek serwera,
Makefile.rkill	plik Makefile zarządzający kompilacją projektu.

Kompilację najprościej jest przeprowadzić z wykorzystaniem programu make[1]:

```
# make -f Makefile.rkill
cc -g  -c -o rkill_clnt.o rkill_clnt.c
cc -g  -c -o rkill_client.o rkill_client.c
cc -g  -o rkill_client  rkill_clnt.o rkill_client.o -lnsl
cc -g  -c -o rkill_svc.o rkill_svc.c
cc -g  -c -o rkill_server.o rkill_server.c
cc -g  -o rkill_server  rkill_svc.o rkill_server.o -lnsl
```

W wyniku kompilacji powinny powstać m.in. dwa programy: rkill_client i rkill_server. Pozwala to na testowe uruchomienie serwera i klienta (np. w różnych oknach środowiska graficznego). Domyślna implementacja serwera nie powoduje jednak wykonania żadnego przetwarzania, stąd trudno zauważyć efekty pracy programu. Obserwację działania serwera umożliwi modyfikacja kodu serwera zawarta w pliku rkill_server.c:

Dodanie linii 8 powoduje wyświetlanie komunikatu na ekranie przy każdorazowym wywołaniu zdalnej procedury. Dalsza modyfikacja kodu serwera umożliwia realizację powierzonego mu zadania:

```
result = kill(pid, 9);
```

Linia ta powinna zostać dopisana na pozycji 9. Argument pid będzie dostarczony przez bibliotekę RPC, a wartość zwrotna funkcji kill() powinna zostać zapisana do statycznej zmiennej result, w celu późniejszego przesłania jej do klienta.

Implementacja kodu klienta wymaga wprowadzenia większej ilości zmian. Przykład 1.3 przedstawia zmodyfikowaną wersję klienta:

W linii 39 dodano odwołanie do drugiego argumentu wywołania programu wraz z konwersją do liczy dziesiętnej (funkcja atoi()). Dodatkowy argument funkcji rkill_prg_1()

[1]Zadaniem programu make jest przeprowadzenie wszystkich niezbędnych kompilacji cząstkowych projektu w odpowiedniej kolejności z zachowaniem zależności pomiędzy plikami źródłowymi.

Przykład 1.2: Przykładowa implementacja serwera

```
1  #include "rkill.h"
2
3  int *
4  rkill_1_svc(int pid,  struct svc_req *rqstp)
5  {
6    static int  result;
7
8    printf("Zdalna procedura\n");
9
10   return &result;
11 }
```

pojawił się również w nagłówku tej funkcji (linia 5). Identyfikator procesu przekazywany jest dalej do pieńka klienta w linii 19. Po każdorazowej zmianie kodu należy oczywiście wykonać ponownie komendę `make`. W obecnej postaci możliwe jest więc zatrzymanie dowolnego procesu na komputerze z uruchomionym serwerem `rkill`:

```
# rkill_client unixlab 1845
```

gdzie `unixlab` jest przykładową nazwą docelowego serwera.

Dalsze modyfikacje kodu aplikacji klienta powinny umożliwić uzyskanie informacji o przebiegu wykonania wysłania sygnału. Należy w tym celu odwołać się do wartości zwracanej przez funkcję `rkill_1` będącą implementacją pieńka klienta. Wartość ta jest wskaźnikiem na liczbę, która przechowuje zwróconą przez serwer liczbę.

Dodatkowy argument funkcji `rkill()`. Jeżeli w wyniku zmiany specyfikacji wymagań należy dodać nowy argument wywołania funkcji `rkill()`, to możliwe są dwa rozwiązania: ręczne korygowanie odpowiednich pieńków i kodów klienta/serwera (niezalecane), lub ponowne wygenerowanie kodu aplikacji (zalecane). Zmodyfikowane wcześniej fragmenty kodu trzeba jednak w drugim przypadku ręcznie wprowadzić do nowej wersji aplikacji. Jako ćwiczenie należy więc zaimplementować usługę o następującym interfejsie:

```
program RKILL_PRG {
  version RKILL_VERSION_1 {
    int rkill(int pid, int sig) = 1;
  } = 1;
} = 0x21000001;
```

Protokół transportowy. Kod programu klienta z przykładu 1.3 zawiera w linii 18 wywołanie funkcji `clnt_create()` tworzącej uchwyt komunikacyjny dla warstwy RPC. Parametrem tej funkcji jest nazwa protokołu transportowego. Zmiana domyślnego protokołu UDP na TCP nie spowoduje zmiany funkcjonalnej aplikacji, co pokazuje niezależność mechanizmu RPC od warstwy transportowej.

Plik Makefile. Nazwę standardowo generowanego przez `rpcgen` pliku `Makefile.rkill` warto zmienić na `Makefile` aby nie używać przełącznika `-f` komendy `make`. Zawartość tego pliku również warto zmienić, ponieważ znajduje się tam wywołanie komendy `rpcgen`

Przykład 1.3: Zmodyfikowany klient usługi rkill

```
#include "rkill.h"


int
rkill_prg_1(char *host, int pid)
{
  CLIENT *clnt;
  int  *result_1;
  int rkill_1_pid;

#ifndef DEBUG
  clnt = clnt_create (host, RKILL_PRG, RKILL_VERSION_1, "udp");
  if (clnt == NULL) {
    clnt_pcreateerror (host);
    exit (1);
  }
#endif  /* DEBUG */

  result_1 = rkill_1(pid, clnt);
  if (result_1 == (int *) NULL) {
    clnt_perror (clnt, "call failed");
  }
#ifndef DEBUG
  clnt_destroy (clnt);
#endif   /* DEBUG */
}


int
main (int argc, char *argv[])
{
  char *host;

  if (argc < 3) {
    printf ("usage: %s server_host\n", argv[0]);
    exit (1);
  }
  host = argv[1];
  rkill_prg_1 (host, atoi(argv[2]));
  exit (0);
}
```

powodujące ponowne tworzenie kodu źródłowego po stwierdzeniu modyfikacji specyfikacji w pliku `rkill.x`. Należy w tym celu usunąć następujące 2 linie:

```
$(TARGETS) : $(SOURCES.x)
    rpcgen $(RPCGENFLAGS) $(SOURCES.x)
```

Drugą zalecaną modyfikacją pliku `Makefile` jest modyfikacja kodu dla operacji `clean`. Standardowo powoduje ona usunięcie kodu źródłowego w języku C! Należy usunąć odwołanie do zmiennej `TARGETS`, uzyskując następujące linie:

```
clean:
    $(RM) core $(OBJECTS_CLNT) $(OBJECTS_SVC) $(CLIENT) $(SERVER)
```

Ostatnia zmiana ma na celu poprawę jakości tworzonego kodu. Zmienna `CFLAGS` ustawiana w pliku `Makefile` umożliwia przesłanie dodatkowych opcji dla kompilatora. Warto włączyć w tym miejscu wypisywanie wszystkich ostrzeżeń:

```
CFLAGS += -g -Wall
```

i doprowadzić kod aplikacji do stanu, w którym kompilacja będzie przebiegała bez żadnych ostrzeżeń.

Zadania

1. Przetestuj działanie klienta usługi `rusers` z przykładu 1.1.
2. Stwórz aplikację `rkill` korzystając z generatora kodu `rpcgen`. Aplikacja klienta powinna być wywoływana z 2 argumentami: nazwą komputera docelowego i identyfikatorem procesu. Serwer powinien wysyłać sygnał SIGINT do wskazanego procesu i zwracać wynik z funkcji `kill()`. Wynik wykonania zdalnej procedury powinien zostać przedstawiony po stronie klienta.
3. Stwórz rozszerzoną aplikację `rkill` udostępniającą dwuargumentową funkcję `rkill()` umożliwiającą wysłanie dowolnego sygnału do dowolnego procesu.
4. Przetestuj działanie aplikacji dla protokołów transportowych UDP i TCP.

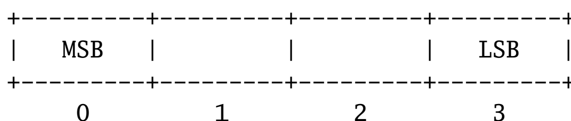
1.4 XDR

1.4.1 Wprowadzenie

XDR (ang. *eXternal Data Representation*) jest standardem reprezentacji danych niezależnym od architektury sprzętu i języków programowania. Jednocześnie XDR dostarcza narzędzi do konwersji danych z lokalnego formatu do formatu niezależnego. XDR został wprowadzony przez firmę Sun Microsystems równolegle z sieciowym systemem plików NFS (zobacz punkt 4). Szczegółowy opis standardu znajduje się w dokumencie RFC 1014 [Mic87].

Podstawowe zasady kodowania XDR to:

- reprezentacja liczb całkowitych jako 32-bitowe ciągi typu *big-endian*:



- reprezentacja liczb rzeczywistych w formacie IEEE (również 4 bajty)
- reprezentacja wszystkich innych typów zajmuje zawsze wielokrotność 4 bajtów (niektóre bajty mogą zostać wypełnione zerami)
- interpretacja ciągów bajtów pozostawiona jest nadawcy i odbiorcy wiadomości — typy nie są kodowane

Pełna dokumentacja funkcji z biblioteki XDR znajduje się na stronie pomocy systemowej `xdr(3)`.

1.4.2 Potoki XDR

Biblioteka XDR składa się ze zbioru funkcji w języku C służących do konwersji podstawowych typów danych do i z standardu XDR. W XDR wyróżnia się następujące dwa pojęcia:

Potok XDR jest to strumień danych zakodowanych zgodnie ze standardem XDR. Istnieją trzy typy potoków XDR: potoki na standardowym wejściu/wyjściu, potoki w pamięci i potoki komunikatów.

Filtr XDR jest to procedura, której zadaniem jest kodowanie/dekodowanie danych określonego typu. Filtry dla podstawowych typów danych są dostarczone z biblioteką XDR. Nowe filtry można konstruować w oparciu o filtry typów prostych.

Potok na standardowym wejściu/wyjściu. Potok taki umożliwia czytanie/pisanie bezpośrednio z pliku. Przykład 1.4 pokazuje kod programu zapisującego do pliku liczbę całkowitą i znak:

W linii 12 tworzony jest potok XDR funkcją `xdrstdio_create()`, której argumentem jest predefiniowana stała `XDR_ENCODE` wskazująca na potok do kodowania (zapisu). W przypadku odczytu należy użyć stałej `XDR_DECODE`. Utworzenie potoku powoduje zainicjowanie struktury `XDR`, która jest uchwytem reprezentującym potok. Funkcje `xdr_*` służą do kodowania/dekodowania potoku w zależności od typu potoku. Po wykonaniu programu można przeanalizować jego zawartość w celu weryfikacji zasad kodowania XDR, np. wykonując poniższą komendę:

```
# hexdump -C /tmp/xdr.test
00000000  00 00 01 02 00 00 00 61                          |.......a|
```

Plik ma więc 8 bajtów. Pierwsze 4 bajty reprezentują liczbę dziesiętną o wartości 258 (1*256+2). Zapis tej samej zmiennej za pomocą funkcji `fwrite()` spowodowałby inne ułożenie bajtów w pliku. Pozostałe 4 bajty pliku reprezentują pojedynczy znak, co powoduje wyzerowanie nieużywanych bajtów.

Przykład 1.4: Potok XDR zapisujący do pliku

```
1  #include <stdio.h>
2  #include <rpc/xdr.h>
3
4  int main()
5  {
6    FILE* f;
7    XDR   xdrh;
8    int   x = 258;
9    char  c = 'a';
10
11   f = fopen("/tmp/xdr.test", "w");
12   xdrstdio_create(&xdrh, f, XDR_ENCODE);
13   xdr_int(&xdrh, &x);
14   xdr_char(&xdrh, &c);
15   fclose(f);
16 }
```

Potok w pamięci. Tworzenie potoku XDR zapisującego dane do bufora w pamięci umożliwia funkcja:

```
void xdrmem_create(XDR* handle, char* addr, int size, xdr_op op);
```

Ten rodzaj potoku ma zastosowanie przy komunikacji procesów poprzez interfejs gniazdek BSD w protokole UDP. Wiadomość przed wysłaniem może być w całości przygotowana w pamięci.

Potok komunikatów. Potok ten umożliwia buforowanie danych przekazywanych między procesami. Potoki komunikatów mają zastosowanie przy komunikowaniu procesów poprzez gniazdka TCP. Do tworzenia potoku służy funkcja o poniższym nagłówku:

```
void xdrrec_create(XDR* handle, int sndSize, int rcvSize, char* io,
                   readProc, writeProc);
```

Argument `sndSize` określa rozmiar bufora nadawczego, a `rcvSize` bufora odbiorczego. Parametr `io` identyfikuje mechanizm komunikacyjny (struktura `FILE`, gniazdko BSD). Parametr `readProc` to nazwa procedury, która jest wywoływana gdy w buforze zabraknie danych do odczytu. Analogicznie `writeProc` jest wywoływana gdy brakuje miejsca w buforze nadawczym. Nagłówek funkcji do obsługi bufora wygląda następująco:

```
int fun(char* io, char* buf, int n);
```

gdzie `n` wskazuje ilość bajtów do przesłania. Funkcja powinna zwrócić ilość faktycznie przesłanych danych.

Korzystanie z potoków komunikatów ułatwiają następujące funkcje:

`xdrrec_endofrecord(XDR* handle, bool_t sendNow)`
: wymuszenie zakończenia komunikatu i jego wysłanie jeżeli parametr `sendNow` ma wartość `TRUE`.

`xdrrec_skiprecord(XDR* handle)`
: przejście do czytania następnego komunikatu z pominięciem pozostałej części komunikatu bieżącego.

`xdrrec_eof(XDR* handle)`
: sprawdzenie dostępności danych do odczytu w buforze odbiorczym.

Inne przydatne funkcje

`int xdr_getpos(XDR* handle)`
: zwraca bieżącą pozycję w potoku.

`bool_t xdr_setpos(XDR* handle, int pos)`
: ustawienie pozycji w potoku.

1.4.3 Filtry XDR

Filtr jest funkcją, której zadaniem jest kodowanie i dekodowanie określonych typów danych. Istnieją filtry proste, złożone i pochodne. Filtry proste umożliwiają kodowanie typów prostych języka C, np. `char`, `int`, np.:

```
bool_t xdr_int(XDR* handle, int* value);
```

Drugim argumentem filtru jest zawsze wskaźnik do danej określonego typu.

Filtry złożone służą do konwersji danych złożonych z typów prostych, a więc np. łańcuchów znaków czy tablic. Obsługiwane są następujące typy danych:

`string`	łańcuch znaków
`opaque`	tablica bajtów o ustalonej wielkości
`bytes`	tablica bajtów o zmiennej wielkości
`vector`	tablica danych dowolnego typu o ustalonej wielkości
`array`	tablica danych dowolnego typu o zmiennej wielkości
`union`	unia
`reference`	wskaźnik
`pointer`	wskaźnik rozpoznający wskaźnik pusty NULL

Filtry pochodne są tworzone w oparciu o inne istniejące filtry. Przykładem może być filtr do kodowania struktury danych.

Zarządzanie pamięcią. Odczytując złożoną strukturę danych odbiorca może nie wiedzieć ile ona będzie zajmować miejsca w pamięci. Alokację pamięci może jednak przeprowadzić automatycznie biblioteka XDR. Oto przykład dla łańcuchów tekstowych:

```
XDR  xdr;
char *buf;
...
buf = NULL;
xdrstdio_create(&xdr, f, XDR_DECODE);
xdr_string(&xdr, &buf, 1024);        /* odczyt napisu z potoku */
...
xdr_free(xdr_string, &buf);
```

Użyta funkcja xdr_free() służy do zwalniania pamięci zaalokowanej przez procedury XDR. Parametr pierwszy tej procedury to wskaźnik na odpowiedni filtr XDR.

1.4.4 Filtry XDR tworzone przez rpcgen

Filtry pochodne XDR mogą być tworzone automatycznie przez generator kodu rpcgen. Wymaga to przygotowania odpowiedniej specyfikacji zbliżonej notacją do składni języka C. W poniższym przykładzie zostanie utworzony filtr do kodowania struktury DANE:

```
struct DANE
{
  int   x;
  char  c;
  float f[10];
};
```

Struktura DANE składa się z pola x typu int, pojedynczego znaku c i 10-elementowej tablicy f przechowującej liczby zmiennoprzecinkowe. Generowanie odpowiednich filtrów realizuje polecenie rpcgen:

```
# rpcgen DANE.x
```

co powoduje powstanie plików DANE.h i DANE_xdr.c zawierających odpowiednio definicje typów na poziomie języka C i implementację filtru XDR. Pliki te zostały przedstawione w przykładach 1.5 i 1.6:

Język programu rpcgen pozwala na definiowanie tablic o zmiennym rozmiarze. Następujące zapisy mogą znaleźć się w specyfikacji struktury danych:

`x[10]` 10-elementowa tablica,

`x<10>` tablica o *maksymalnym* rozmiarze równym 10,

`x<>` tablica o nieokreślonym rozmiarze.

Wykorzystanie tablic o dynamicznym rozmiarze powoduje wygenerowanie struktur danych języka C dla każdej dynamicznej tablicy. W przypadku wspomnianej struktury DANE jeżeli tablica f miałaby określony tylko maksymalny rozmiar to definicja typu w języku C wyglądałaby następująco:

```
struct DANE {
  int x;
  char c;
  struct {
    u_int f_len;
    float *f_val;
  } f;
};
```

W strumieniu XDR w takim przypadku przed wypisaniem zawartości tablicy zostanie najpierw umieszczony jej rozmiar.

Nowe typy danych oraz stałe definiowane są na poziomie specyfikacji dla rpcgen podobnie jak w języku C:

```
const MAX = 100;

typedef int Tablica<MAX>;
```

Przykład 1.5: Plik nagłówkowy dla filtru XDR

```
1  #ifndef _DANE_H_RPCGEN
2  #define _DANE_H_RPCGEN
3
4  #include <rpc/rpc.h>
5
6
7  #ifdef __cplusplus
8  extern "C" {
9  #endif
10
11
12 struct DANE {
13   int x;
14   char c;
15   float f[10];
16 };
17 typedef struct DANE DANE;
18
19 /* the xdr functions */
20
21 #if defined(__STDC__) || defined(__cplusplus)
22 extern  bool_t xdr_DANE (XDR *, DANE*);
23
24 #else /* K&R C */
25 extern bool_t xdr_DANE ();
26
27 #endif /* K&R C */
28
29 #ifdef __cplusplus
30 }
31 #endif
32
33 #endif /* !_DANE_H_RPCGEN */
```

Przykład 1.6: Implementacja filtru XDR

```
1  #include "DANE.h"
2
3  bool_t
4  xdr_DANE (XDR *xdrs, DANE *objp)
5  {
6    register int32_t *buf;
7
8    int i;
9     if (!xdr_int (xdrs, &objp->x))
10      return FALSE;
11    if (!xdr_char (xdrs, &objp->c))
12      return FALSE;
13    if (!xdr_vector (xdrs, (char *)objp->f, 10,
14     sizeof (float), (xdrproc_t) xdr_float))
15      return FALSE;
16   return TRUE;
17 }
```

Wskaźniki. Definicja typów dla programu `rpcgen` pozwala na definiowanie wskaźników, które będą następnie poprawnie odtwarzane przy odczycie. Wspomniana struktura `DANE` mogłaby więc być uzupełniona o wskazanie na następny element:

```
struct DANE {
  int x;
  DANE* next;
};
```

Do kodowania wskaźników wykorzystywany jest filtr `xdr_pointer()`.

Zadania

1. Napisz program dekodujący dane zapisane w pliku z przykładu 1.4.
2. Sprawdź w jaki sposób odbywa się binarne kodowanie łańcuchów znaków.
3. Napisz program zapisujący do pliku i odczytujący strukturę o następujących polach: liczba typu `int`, tablica znaków o długości 5, liczba zmiennoprzecinkowa.
4. Napisz program, który zapisze do pliku listę rekordów przechowujących liczby typu `int`. Zdefiniuj w tym celu strukturę zawierającą wskaźnik na następny rekord. Sprawdź zachowanie biblioteki XDR w przypadku zapisu listy cyklicznej.
5. Napisz program klienta i serwera aplikacji udostępniającej informacje o pliku: rozmiar pliku, identyfikator właściciela, daty modyfikacji i dostępu, itp.
6. Zaproponuj implementację udoskonalonej funkcji `xdr_pointer()` obsługującej dowolnie złożone (a więc również i cykliczne) dynamiczne struktury danych.

1.5 Asynchroniczne RPC

1.5.1 Przebieg zdalnego wywołania

Biblioteka RPC po wywołaniu zdalnej procedury oczekuje standardowo około 25 sekund na odpowiedź. Po upływie tego czasu pieniek klienta zgłasza wystąpienie błędu przekroczenia czasu oczekiwania (*timeout*) pomimo, że odpowiedź nadchodzi. Można to zweryfikować wstawiając do implementacji zdalnej metody wywołanie funkcji:

```
sleep(30);
```

powodującej wydłużenie czasu realizacji zdalnego wywołania. Wykonanie programu klienta kończy się w takiej sytuacji błędem:

```
# rkill_client localhost 2314
call failed: RPC: Timed out
```

Błąd wynika oczywiście z przekroczenia czasu oczekiwania na odpowiedź. Ciekawa jest jednak reakcja serwera:

```
# rkill_server
Zdalna procedura
Zdalna procedura
Zdalna procedura
Zdalna procedura
Zdalna procedura
```

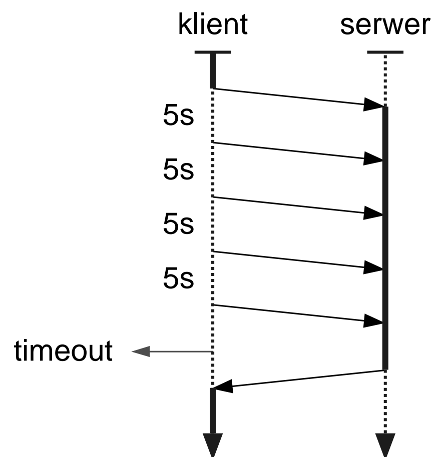
Okazuje się, że zdalna procedura została wywołana aż 5 razy. Jest to wynik strategii stosowanej podczas korzystania z bezpołączeniowego protokołu transportowego UDP. Wysłanie komunikatu UDP do serwera i brak odpowiedzi po upływie czasu określonego parametrem `RETRY_TIMEOUT` powoduje wysłanie kolejnego komunikatu (zobacz rys. 1.3). Domyślna wartość parametry `RETRY_TIMEOUT` wynosi 5 sekund, co powoduje, że w przeciągu 25 sekund klient zdąży wysłać 5 komunikatów z żądaniem. Po upływie czasu określanego parametrem `TIMEOUT` następuje zasygnalizowanie błędu warstwie wyższej, czyli aplikacji. Serwer obsługuje poszczególne żądania sekwencyjnie, ale warstwa komunikacyjna systemu operacyjnego odbiera przesyłane komunikaty i buforuje je, co powoduje ich późniejsze wykonanie.

Inne zachowanie biblioteki RPC można zaobserwować gdy protokołem transportowym będzie protokół TCP. W takiej sytuacji klient nie retransmituje żądania, ponieważ ma gwarancję, że komunikat dotarł do serwera. Błąd przekroczenia czasu spowoduje przerwanie zdalnego wywołania, ale serwer odbierze jedynie pojedyncze żądanie.

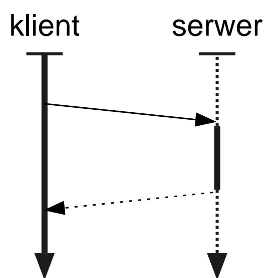
1.5.2 Proste wywołanie asynchroniczne

Jeżeli nie można określić maksymalnego czasu wykonania zdalnej procedury, pozostaje wykonanie wywołania asynchronicznego, czyli takiego, w którym serwer ma dowolnie dużo czasu na realizację żądania. Asynchronizm powoduje również, że klient nie jest blokowany realizacją zdalnego wywołania i może kontynuować swoje przetwarzanie. Rysunek 1.4 przedstawia schemat komunikacji w przypadku wywołania asynchronicznego.

Oczywistą konsekwencją wywołania asynchronicznego jest niemożliwość przekazania wyniku zdalnej procedury do klienta. Oznacza to, że definicja usługi z punktu 1.3.1 musi zostać zmieniona do postaci:



Rys. 1.3: Schemat komunikacji z serwerem po protokole UDP dla długotrwałych procedur



Rys. 1.4: Wywołanie asynchroniczne

```
program RKILL_PRG {
  version RKILL_VERSION_1 {
    void rkill(int pid) = 1;
  } = 1;
} = 0x21000001;
```

Najprostszą realizacją asynchronicznych wywołań jest modyfikacja domyślnego czasu oczekiwania na odpowiedź. Czas oczekiwania klienta jest zdefiniowany w nagłówku pieńka klienta poprzez statyczną strukturę `TIMEOUT`:

```
static struct timeval TIMEOUT = { 25, 0 };
```

W strukturze `timeval` pole `tv_sec` określa liczbę sekund, a pole `tv_usec` liczbę mikrosekund (zobacz np. stronę pomocy systemowej `utimes(3)`). Czas ten można również zmodyfikować w pliku implementacyjnym klienta korzystając z funkcji `clnt_control()`[2]. W przypadku przykładu 1.3 należy w tym celu zmienić plik `rkill_client.c` w miejscu, gdzie jest tworzony uchwyt komunikacyjny:

```
struct timeval tm = { 60, 0 };
...
clnt = clnt_create (host, RKILL_PRG, RKILL_VERSION_1, "udp");
...
clnt_control(clnt, CLSET_TIMEOUT, &tm);
```

Szczególną zmianą czasu oczekiwania jest ustawienie go na wartość 0, co w praktyce oznacza ignorowanie odpowiedzi i permanentne zgłaszanie błędu *timeout*. Błąd przekroczenia czasu oczekiwania powinien oczywiście być ignorowany:

```
struct rpc_err err;
...
result_1 = rkill_1(rkill_1_pid, clnt);
if (result_1 == NULL) {
  clnt_geterr(clnt, &err);
  if (err.re_status != RPC_TIMEDOUT) {
    clnt_perror (clnt, "call failed");
  }
}
```

Funkcja `clnt_geterr()` umożliwia pobranie szczegółowych informacji o zaistniałym błędzie[3]. Struktura `rpc_err` posiada pole `re_status`, które przechowuje kod zaistniałego błędu. W powyższym przykładzie wyświetlana będzie więc informacja o wszystkich błędach oprócz `RPC_TIMEDOUT`.

Przykład 1.7 zawiera pełen kod implementacji klienta wykorzystującego asynchroniczne wywołanie:

1.5.3 Modyfikacja pieńka serwera

Ciekawą, choć niezalecaną oficjalnie implementacją asynchronicznego wywołania procedury jest modyfikacja pieńka serwera tak, aby wysyłał odpowiedź na żądanie klienta *zanim* wykona zdalną procedurę. Zabezpiecza to oczywiście klienta przed błędem typu *timeout*

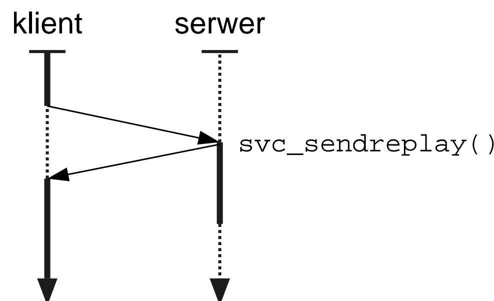
[2] Funkcja `clnt_control()` umożliwia również zmianę parametru `RETRY_TIMEOUT`.

[3] Jest to analogia do zmiennej globalnej `errno` przechowującej kod błędu ostatnio wywołanej funkcji systemowej. Zobacz również stronę pomocy systemowej `errno(3)`.

Przykład 1.7: Implementacja klienta wykorzystującego asynchroniczne wywołanie procedury

```c
#include "rkill.h"

void
rkill_prg_1(char *host, int pid)
{
  CLIENT *clnt;
  void  *result_1;
  struct timeval tm = { 0, 0 };
  struct rpc_err err;

#ifndef DEBUG
  clnt = clnt_create (host, RKILL_PRG, RKILL_VERSION_1, "udp");
  if (clnt == NULL) {
    clnt_pcreateerror (host);
    exit (1);
  }
  clnt_control(clnt, CLSET_TIMEOUT, (char*)&tm);
#endif  /* DEBUG */

  result_1 = rkill_1(pid, clnt);
  if (result_1 == (void *) NULL) {
    clnt_geterr(clnt, &err);
    if (err.re_status != RPC_TIMEDOUT) {
      clnt_perror (clnt, "call failed");
    }
  }
#ifndef DEBUG
  clnt_destroy (clnt);
#endif   /* DEBUG */
}


int
main (int argc, char *argv[])
{
  char *host;

  if (argc < 3) {
    printf ("usage: %s server_host pid\n", argv[0]);
    exit (1);
  }
  host = argv[1];
  rkill_prg_1 (host, atoi(argv[2]));
  exit (0);
}
```

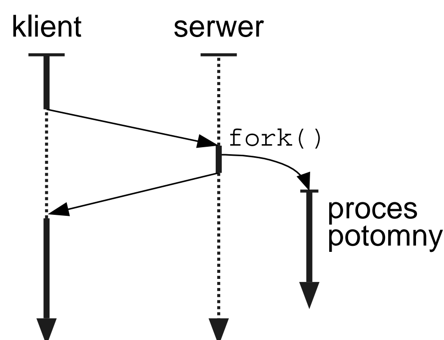
i dodatkowo daje mu potwierdzenie poprawnego wywołania procedury po stronie serwera (zobacz rys. 1.5). Podobnie jednak jak w przykładzie z zerowym czasem oczekiwania na odpowiedź nie będzie możliwe przesłanie wyniku procedury. Przykład 1.8 pokazuje zmodyfikowaną procedurę pieńka serwera. Wewnątrz instrukcji `switch` (od linii 26) następuje zainicjowanie zmiennych reprezentujących filtry kodujące argument i wynik zdalnej procedury oraz zmiennej wskazującej na procedurę, która ma być wywołana. Linia 50 zawiera wywołanie zdalnej procedury. Wcześniej jednak (linia 46) następuje odesłanie odpowiedzi. Jako wynik przetwarzania przesyłany jest pusty wskaźnik NULL. W przypadku usług, które udostępniają większą liczbę funkcji obsługę asynchronicznych wywołań należałoby przenieść do wnętrza instrukcji `switch`.



Rys. 1.5: Wywołanie asynchroniczne z wczesnym odesłaniem odpowiedzi

1.5.4 Procesy potomne

Ostatnia propozycja realizacji wywołania asynchronicznego polega na wykorzystaniu procesów potomnych do realizacji samej procedury. Ponieważ proces główny serwera będzie zajmował się jedynie odbiorem żądań i tworzeniem nowych procesów, jego odpowiedź będzie trafiała natychmiast do klientów (zobacz rys. 1.6). Całe przetwarzanie (potencjalnie długie) odbędzie się w nowym procesie potomnym. Umożliwia to współbieżne wykonywanie wielu zdalnych procedur przez pojedynczy serwer. Przykład 1.9 prezentuje implementację zdalnej procedury takiego serwera. Pełen kod serwera musi uwzględnić całościowo problem zarządzania procesami potomnymi, a więc m.in. problem zarządzania procesami *zombie*.



Rys. 1.6: Wywołanie asynchroniczne z procesem potomnym

Procesy potomne lub przetwarzanie wielowątkowe[4] można wykorzystać również w przypadku synchronicznego wywołania procedur w celu zrównoleglenia przetwarzania po stro-

[4]Zobacz punkt 1.8.

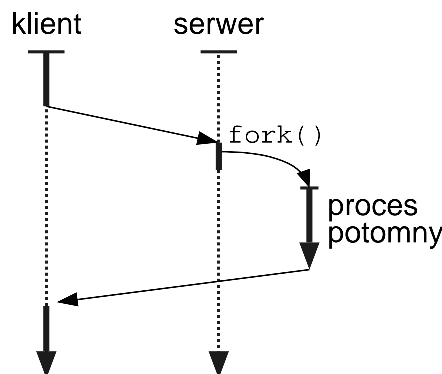
Przykład 1.8: Implementacja procedury z pieńka serwera dla asynchronicznego wywołania z wczesnym odesłaniem odpowiedzi (bez funkcji `main()`)

```
#include "rkill.h"
#include <stdio.h>
#include <stdlib.h>
#include <rpc/pmap_clnt.h>
#include <string.h>
#include <memory.h>
#include <sys/socket.h>
#include <netinet/in.h>

static void *
_rkill_1 (int  *argp, struct svc_req *rqstp)
{
  return (rkill_1_svc(*argp, rqstp));
}

static void
rkill_prg_1(struct svc_req *rqstp, register SVCXPRT *transp)
{
  union {
    int rkill_1_arg;
  } argument;
  char *result;
  xdrproc_t _xdr_argument, _xdr_result;
  char *(*local)(char *, struct svc_req *);

  switch (rqstp->rq_proc) {
  case NULLPROC:
    (void) svc_sendreply (transp, (xdrproc_t) xdr_void, (char *)NULL);
    return;

  case rkill:
    _xdr_argument = (xdrproc_t) xdr_int;
    _xdr_result = (xdrproc_t) xdr_void;
    local = (char *(*)(char *, struct svc_req *)) _rkill_1;
    break;

  default:
    svcerr_noproc (transp);
    return;
  }
  memset ((char *)&argument, 0, sizeof (argument));
  if (!svc_getargs (transp, (xdrproc_t) _xdr_argument, (caddr_t) &argument)) {
    svcerr_decode (transp);
    return;
  }
  if (!svc_sendreply(transp, (xdrproc_t) _xdr_result, NULL)) {
    svcerr_systemerr (transp);
  }
  else {
    result = (*local)((char *)&argument, rqstp);
  }
  if (!svc_freeargs (transp, (xdrproc_t) _xdr_argument, (caddr_t) &argument)) {
    fprintf (stderr, "%s", "unable to free arguments");
    exit (1);
  }
  return;
}
```

Przykład 1.9: Implementacja zdalnej procedury korzystająca z procesu potomnego

```
1  #include "rkill.h"
2
3  void *
4  rkill_1_svc(int pid,  struct svc_req *rqstp)
5  {
6    if (fork()==0)
7    {
8      printf("Zatrzymywanie procesu %d.\n", pid);
9      kill(pid, 9);
10     sleep(30);
11     exit(0);
12   }
13
14   return 1;
15 }
```

nie serwera i jednocześnie minimalizacji czasu odpowiedzi dla krótkich żądań. Wymaga to modyfikacji pieńka serwera w celu utworzenia nowego procesu (wątku) dla każdego wywołania zdalnej procedury. Schemat przetwarzania zdalnego wywołania procedury w takim podejściu został zaprezentowany na rys. 1.7.



Rys. 1.7: Wywołanie synchroniczne z procesem potomnym

Zadania

1. Przetestuj obsługę zdalnego wywołania metody `rkill()` po dodaniu do jej implementacji opóźnienia 30-sekundowego.

2. Sprawdź obsługę zdalnego wywołania w przypadku użycia protokołu transportowego TCP.

3. Zaimplementuj wywołanie asynchroniczne poprzez ustawienie zerowego czasu odczekiwania na odpowiedź. Aplikacja klienta nie powinna sygnalizować żadnego błędu, jeżeli przesłanie żądanie udaje się przesłać.

4. Zmodyfikuj pieniek serwera standardowej implementacji zdalnego wywołania metody (z domyślnym czasem oczekiwania na odpowiedź), tak aby odsyłał potwierdzenie

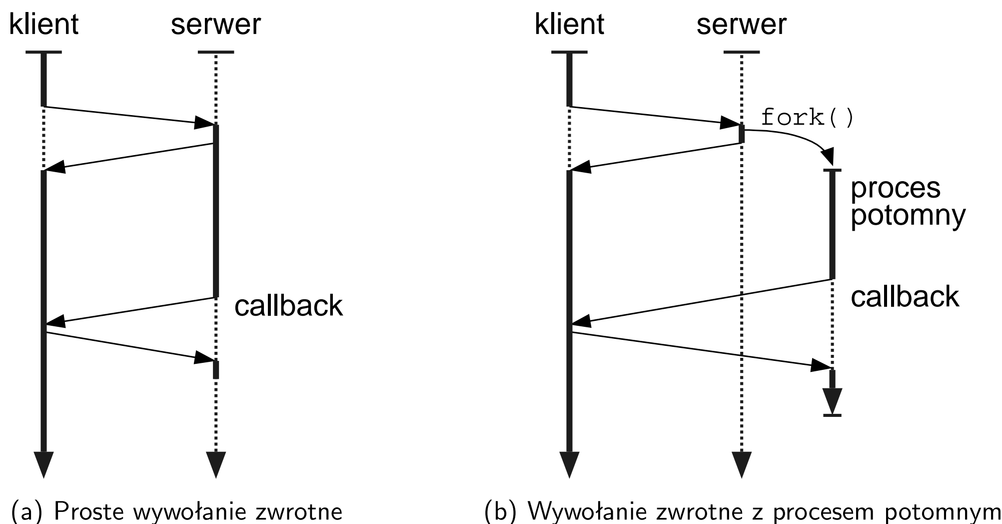
wykonania metody zaraz po odebraniu żądania. Sprawdź jak realizowana będzie teraz współbieżna obsługa długich, 30-sekundowych żądań pochodzących od dwóch różnych klientów.

5. Zmodyfikuj implementację z poprzedniego punktu przenosząc obsługę żądań klientów do procesów potomnych. Ponownie zweryfikuj obsługę współbieżnych żądań.

1.6 Wywołanie zwrotne

1.6.1 Wprowadzenie

Omówione dotąd modele wywołań zdalnych nie dają możliwości odbioru wyniku działania procedury zdalnej w przypadku dowolnie długotrwałego przetwarzania. Zwrócenie wyniku może jednak zostać zrealizowane jako wywołanie zwrotne, w którym serwer, po zakończeniu przetwarzania, wywołuje procedurę zdalną klienta. Rysunek 1.8a pokazuje przykład realizacji takiego wywołania. W wywołaniu zwrotnym klient staje się na pewien czas serwerem, aby móc odebrać wynik zdalnej procedury. Istotną zaletą wywołania zwrotnego jest możliwość nieprzerwanej pracy klienta. Z reguły wymaga to jednak uruchomienia dodatkowego wątku, którego zadaniem będzie oczekiwanie na odbiór wyniku z serwera.



(a) Proste wywołanie zwrotne

(b) Wywołanie zwrotne z procesem potomnym

Rys. 1.8: Wywołanie zwrotne

Wywołanie zwrotne można oczywiście łączyć z obsługą żądań w procesach potomnych co zobrazowano na rysunku 1.8b. Takie podejście zostanie zaprezentowane w następnych przykładach.

1.6.2 Przykład aplikacji

Jako przykład rozważmy znów serwer `rkill` z punktu 1.3.1:

```
program RKILL_PRG {
  version RKILL_VERSION_1 {
    void rkill(int pid, int sig) = 1;
  } = 1;
} = 0x20000000;
```

Zwróćmy uwagę, że funkcja `rkill()` nie zwraca żadnej wartości. Definicję serwera należy uzupełnić o definicję wywołania zwrotnego klienta (plik `rkillcb.x`):

```
program RKILL_CB_PRG {
  version RKILL_CB_VERSION_1 {
    void sendresult(int res) = 1;
  } = 1;
} = 0x40000000;
```

Serwer ten definiuje funkcję `sendresult()`, której zadaniem będzie przesłanie wyniku przetwarzania do klienta. Funkcja ta również nie zwraca żadnego wyniku.

Definicje serwerów należy przetworzyć generatorem kodu `rpcgen`:

```
# rpcgen -N -a rkill.x
# rpcgen -N -a rkillcb.x
```

Oto lista modyfikacji jakie należy wprowadzić do wygenerowanych plików:

1. Usunięcie pliku `Makefile.rkillcb`. Całość sterowania kompilacją znajdzie się w pliku `Makefile.rkill`.
2. Modyfikacja pliku `Makefile.rkill`. Należy uzupełnić listę modułów programowych wchodzących w skład kodu klienta i serwera dopisując odpowiednie wartości do zmiennych `TARGETS_SVC.c` i `TARGETS_CLNT.c`:

   ```
   TARGETS_SVC.c = ... rkillcb_clnt.c rkillcb_client.c
   TARGETS_CLNT.c = ... rkillcb_svc.c rkillcb_server.c
   ```

   Powyższa modyfikacja wskazuje na to, że aplikacja klienta będzie się składać z właściwego klienta (m.in. pliki `rkill_client.c` i `rkill_clnt.c`) oraz z serwera wywołania zwrotnego (pliki `rkillcb_server.c` i `rkillcb_svc.c`). Analogicznie w skład aplikacji serwera wejdzie dodatkowo klient wywołania zwrotnego (pliki `rkillcb_client.c` i `rkillcb_clnt.c`).
3. Przekazywanie argumentów po stronie klienta (plik `rkill_client.c`). Należy odczytać argumenty z linii poleceń i przekazać je poprzez funkcję `rkill_prg_1()` aż do pieńka klienta czyli do funkcji `rkill_1()`.
4. Modyfikacja klienta wywołania zwrotnego (plik `rkillcb_client.c`): całkowite usunięcie definicji funkcji `main()`, zmiana nazwy funkcji `rkill_cb_prg_1()` na `rkill_cb_1()`[5], wyprowadzenie argumentu `res` na zewnątrz i dodanie deklaracji tej funkcji do pliku nagłówkowego `rkillcb.h`. Funkcja `main()` dla serwera zdefiniowana jest w pliku `rkill_svc.c`.
5. Implementacja właściwego serwera (przedstawiona w przykładzie 1.10). Uwaga: istotne jest dołączenie odpowiednich plików nagłówkowych.
6. Implementacja serwera wywołania zwrotnego. Zadaniem serwera jest odbiór wyniku zdalnej procedury i wyświetlenie go na ekranie.
7. Zmiana nazwy funkcji `main()` w pieńku serwera wywołania zwrotnego (plik `rkillcb_svc.c`) na `main2()` i dodanie jej deklaracji do pliku nagłówkowego `rkillcb.h`. Definicja funkcji `main()` dla aplikacji klienta znajduje się w pliku `rkill_client.c`.

[5]Dla uniknięcia konfliktu z funkcją o tej samej nazwie z pieńka serwera wywołania zwrotnego.

Przykład 1.10: Implementacja serwera z wywołaniem zwrotnym

```
1  #include <unistd.h>
2  #include <signal.h>
3  #include <arpa/inet.h>
4  #include "rkill.h"
5  #include "rkillcb.h"
6
7  void *
8  rkill_1_svc(int pid, int sig, struct svc_req *rqstp)
9  {
10   static char * result = NULL;
11
12   printf("rkill(%d,%d)\n", pid, sig);
13   if (fork()==0)
14   {
15     int res;
16     sleep(3);
17     res = kill(pid, sig);
18     rkill_cb_1(inet_ntoa(rqstp->rq_xprt->xp_raddr.sin_addr), res);
19     printf("rkill(%d,%d)  ==>   %d\n", pid, sig, res);
20     exit(0);
21   }
22
23   return (void *) &result;
24 }
```

8. Rozszerzenie implementacji właściwego klienta. Klient po wywołaniu zdalnej procedury (`rkill_prg_1()`) może po prostu wykonać kod serwera wywołania zwrotnego (funkcja `main2()` z pliku `rkillcb_svc.c`).

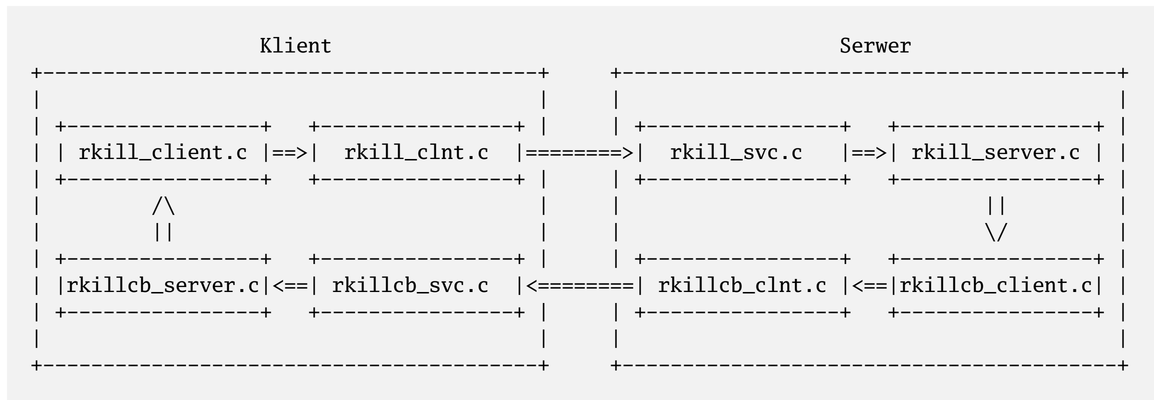
Przeprowadzenie powyższych zmian może wymagać modyfikacji plików nagłówkowych oraz ich dołączania do implementacji poszczególnych fragmentów kodu.

Przepływ sterowania w ramach poszczególnych plików składowych aplikacji został przedstawiony na rysunku 1.9.

1.6.3 Procedura svc_run()

Po wprowadzeniu powyższych modyfikacji klient spowoduje wywołanie zdalnej procedury, serwer wykona ją, wywoła procedurę zwrotną po stronie klienta, ale przetwarzanie po stronie klienta nie zakończy się. Wynika to z charakteru pracy standardowego serwera, którego zadaniem jest praca w pętli nieskończonej. Implementacja klienta korzysta ze standardowego serwera (funkcja `main2()` w pliku `rkillcb_svc.c`). Serwer RPC wywołuje funkcję `svc_run()`, której zadaniem jest nieskończone oczekiwanie i obsługa żądań RPC. Klient korzystający z wywołania zwrotnego powinien oczekiwać tylko na jedno wywołanie RPC od serwera, co wymaga zmodyfikowania funkcji `svc_run()` przedstawionej w przykładzie 1.11.

Procedura `svc_run2()` monitoruje funkcją `select()` deskryptory wskazane zmienną `svc_fdset` i po stwierdzeniu odbioru nowego żądania wywołuje funkcję `svc_getreqset()` przetwarzającą wywołanie. Zewnętrzna zmienna `done` informuje procedurę czy nastąpiło już poprawne wywołanie procedury, na którą oczekuje klient. W linii 23 następuje wyjście



Rys. 1.9: Przepływ sterowania w wywołaniu zwrotnym RPC

Przykład 1.11: Zmodyfikowana procedura svc_run()

```
1  #include <rpc/rpc.h>
2  #include <errno.h>
3  #include "rkillcb.h"   /* tu znajduje sie deklaracja zmiennej done */
4
5  void svc_run2()
6  {
7    fd_set readfds;
8
9    for (;;)
10   {
11     readfds = svc_fdset;
12     switch (select(_rpc_dtablesize(), &readfds, (fd_set*)NULL, (fd_set*)NULL,
13              (struct timeval*)NULL))
14     {
15       case -1:
16         if (errno == EINTR) continue;
17         perror("svc_run: - select failed");
18         return;
19       case 0:
20         continue;
21       default:
22         svc_getreqset(&readfds);
23         if (done) return;
24     }
25   }
26 }
```

Przykład 1.12: Dynamiczna rejestracja usługi

```
1  int register_tmp(SVCXPRT *transp)
2  {
3    long prognum = 0x40000000;
4
5    while(pmap_set(prognum, RKILL_CB_VERSION_1, IPPROTO_UDP, transp->xp_port)==0)
6    {
7      prognum++;
8    }
9    if (!svc_register(transp, prognum, RKILL_CB_VERSION_1,
10                     rkill_cb_prg_1, IPPROTO_UDP))
11   {
12     fprintf (stderr, "%s", "unable to register transient procedure.");
13     exit(1);
14   }
15   return prognum;
16 }
```

z procedury `svc_run2()` po poprawnym obsłużeniu wywołania RPC. Pieniek serwera wywołania zwrotnego (plik `rkillcb_svc.c`) powinien więc wywoływać funkcję `svc_run2()` zamiast standardowej `svc_run()`. Zmienna `done` powinna być inicjowana wewnątrz aplikacji klienta (plik `rkill_client.c`) i modyfikowana w implementacji procedury zwrotnej (plik `rkillcb_server.c`). Wymaga to dodania deklaracji tej zmiennej do pliku nagłówkowego `rkillcb.h` i definicji do kodu klienta. Poprawna kompilacja kodu klienta wymaga również dołączenia definicji funkcji `svc_run2()` do kodu klienta, co oznacza konieczność wskazania w pliku `Makefile` nazwy pliku z implementacją:

```
TARGETS_CLNT.c = ... svc_run2.c
```

Deklaracja funkcji `svc_run2()` powinna również trafić do pliku nagłówkowego `rkillcb.h`.

Funkcja `svc_run()` standardowo nie kończy się nigdy, dlatego pieniek serwera wyświetla błąd po jej zakończeniu. Komunikat ten, jak również i występujące po nim zakończenie procesu funkcją `exit()`, należy oczywiście usunąć.

1.6.4 Rejestracja tymczasowej usługi

Klient uruchamiając lokalny serwer musi go zarejestrować w usłudze `portmap`. Rejestracja powinna odbywać się z użyciem tymczasowego numeru usługi RPC, gdyż w przeciwnym wypadku może wystąpić konflikt pomiędzy użytkownikami uruchamiającymi swoje aplikacje na jednym komputerze. Rejestracji usługi dokonuje pieniek serwera zawarty w pliku `rkillcb_svc.c` (funkcja `main2()` z punktu 1.6.2). Przykład 1.12 prezentuje funkcję `register_tmp()` rejestrującą tymczasowy serwer RPC. Jest to zmodyfikowana wersja funkcji `main()` standardowego pieńka serwera. Funkcja `pmap_set()` jest odpowiedzialna za rejestrację usługi RPC w usłudze `portmap`. Pętla `while` w linii 5 dokonuje próbnej rejestracji pod kolejnymi numerami począwszy od 0x40000000.

Aplikacja klienta przed wywołaniem zdalnej metody powinna zarejestrować procedurę RPC do wywołania zwrotnego, następnie oczekiwać w funkcji `svc_run2()` i na końcu wyrejestrować tymczasowo zarejestrowaną usługę.

Numer tymczasowo zarejestrowanej usługi musi trafić do serwera, aby mógł on przekazać zwrotnie wynik przetwarzania do właściwego klienta. Najprościej można to zrobić

podczas samego wywołania zdalnej procedury. Definicja usługi `rkill` musi więc ulec modyfikacji:

```
program RKILL_PRG {
  version RKILL_VERSION_1 {
    void rkill(int prognum, int pid, int sig) = 1;
  } = 1;
} = 0x20000000;
```

Wymagana będzie więc zmiana implementacji klienta wywołania zwrotnego (plik `rkillcb_client.c`) w celu przekazania mu właściwego numeru usługi RPC. Ostateczna implementacja funkcji `main()` w aplikacji klienta wygląda więc następująco:

```
int
main (int argc, char *argv[])
{
  char *host;
  int prognum;
  int pid;
  int sig;

  if (argc < 4) {
    printf ("usage: %s server_host pid sig\n", argv[0]);
    exit (1);
  }
  host = argv[1];
  pid = atoi(argv[2]);
  sig = atoi(argv[3]);
  done = 0;

  SVCXPRT *transp = svcudp_create(RPC_ANYSOCK);
  if (transp == NULL)
  {
    fprintf (stderr, "%s", "cannot create udp service.");
    exit(1);
  }
  prognum = register_tmp(transp);
  rkill_prg_1(host, prognum, pid, sig);
  svc_run2();
  svc_unregister(prognum, RKILL_CB_VERSION_1);
  svc_destroy(transp);
  exit (0);
}
```

Funkcja `svc_unregister()` wyrejestrowuje tymczasową usługę RPC, a funkcja `svc_destroy()` zwalnia zasoby uchwytu komunikacyjnego.

Dla poprawnej kompilacji projektu deklaracja funkcji `register_tmp()` powinna zostać dołączona do pliku nagłówkowego `rkillcb.h`.

Zadania

1. Zweryfikuj poprawność wykonania aplikacji klienta w przypadku współbieżnego uruchamiania wielu kopii na jednym komputerze.
2. Zrealizuj wielowątkową implementację klienta z wywołaniem zwrotnym. Zadaniem dodatkowego wątku ma być oczekiwanie na odpowiedź z serwera.

3. Zrealizuj aplikację rozpraszającą obliczenia na N komputerów z wykorzystaniem RPC. Komputery obliczeniowe powinny udostępniać zdalne wywołanie wykonujące fragment obliczeń i zwracające wyniki poprzez wywołanie zwrotne. Aplikacja klienta prezentuje ostateczny wynik po odebraniu wyników cząstkowych od wszystkich N serwerów.

1.7 Kontrola praw dostępu

Zdalne wywołania mogą być uzupełnione o informację identyfikującą użytkownika. Poniższy fragment kodu powinien zostać umieszczony w implementacji klienta:

```
clnt = clnt_create(...);
...
clnt->cl_auth = authunix_create_default();
```

Funkcja `authunix_create_default()` powoduje dołączenie do wywołania procedury informacji o nazwie komputera klienta, identyfikatorze użytkownika (UID) i identyfikatorze jego grupy (GID). Po stronie serwera informacje te dostępne są poprzez strukturę `svc_req` przekazywaną do zdalnej struktury:

```
struct authunix_parms *aup;
aup = rqstp->rq_clntcred;
printf("Komputer: %s\n", aup->aup_machname);
printf("UID     : %d\n", aup->aup_uid);
printf("GID     : %d\n", aup->aup_gid)
```

1.8 Wątki w standardzie POSIX

1.8.1 Wprowadzenie

Wiele nowoczesnych systemów operacyjnych umożliwia dekompozycję złożonych zadań nie tylko na rozłączne, współpracujące ze sobą procesy, ale również na wątki. Wątek w systemie operacyjnym to niezależny strumień przetwarzania pracujący w ramach jednego procesu z innymi wątkami. Wątki współdzielą między sobą przestrzeń adresową procesu, mają więc dostęp do tych samych danych statycznych i dynamicznych, ale posiadają swój własny stos umożliwiający im niezależne wykonywanie procedur. Poniższy punkt zawiera opis interfejsu programistycznego w standardzie POSIX. Więcej informacji można znaleźć w pracy [Gra98] oraz dokumentacji [Fre01].

Przykład 1.13 prezentuje aplikację tworzącą nowy wątek realizujący proste przetwarzanie. Kluczowym elementem programu jest wywołanie funkcji `pthread_create()` tworzącej nowy wątek:

```
int pthread_create(pthread_t * thread,
                   pthread_attr_t * attr,
                   void * (*start_routine)(void *),
                   void * arg);
```

Nowy wątek powstaje jako współbieżne wywołanie istniejącej w programie procedury. Procedura taka powinna pobierać wskaźnik `void*` jako argument i zwracać taką wartość

Przykład 1.13: Przykład programu tworzącego nowy wątek

```
1  #include <stdio.h>
2  #include <unistd.h>
3  #include <pthread.h>
4  #include <string.h>
5
6  void* work(void* arg)
7  {
8    printf("Praca wątku\n");
9    sleep(3);
10   return NULL;
11 }
12
13 int main()
14 {
15   pthread_t th;
16
17   printf("Tworzenie wątku...\n");
18   pthread_create(&th, NULL, work, NULL);
19   printf("Wątek uruchomiony...\n");
20   pthread_join(th, NULL);
21   printf("Wątek zakończony.\n");
22   return 0;
23 }
```

jako wynik przetwarzania. Argument funkcji implementującej wątek umożliwia parametryzowanie wątków. Aktualna wartość Parametru przekazywana jest ostatnim argumentem wywołania `pthread_create()`. Poprawne utworzenie wątku inicjuje zmienną typu `pthread_t`, która jest liczbą typu `unsigned int`. Wartość ta staje się unikalnym w ramach procesu identyfikatorem wątku. Wskaźnik do struktury `pthread_attr_t` umożliwia zainicjowanie pewnych atrybutów wątku (m.in. rozmiar stosu, algorytm szeregowania, priorytet).

Wszystkie funkcje implementujące obsługę wątków mają nazwy zaczynające się od `pthread_` i są udokumentowane w sekcji `3p` standardowej pomocy systemowej. Każdy program korzystający z wątków powinien dołączyć plik nagłówkowy `pthread.h`. Kompilacja programu korzystającego z wątków wymaga dołączenie biblioteki `pthread`:

```
# gcc -o thread thread.c -lpthread
```

W systemie Linux wątki POSIX są implementowane na poziomie systemu operacyjnego. Ich obecność można zaobserwować korzystając np. z opcji `H` standardowej komendy `ps`:

```
# ./thread &
# ps lxH
F  UID   PID  PPID PRI  NI    VSZ   RSS WCHAN  STAT TTY     TIME COMMAND
...
0  501  6446  6445  15   0   4360  1936 wait   Ss   pts/2   0:00 /bin/bash
0  501  6457  6446  15   0   5652   460 347332 Sl   pts/2   0:00 ./thread
1  501  6457  6446  16   0   5652   460 -      Sl   pts/2   0:00 ./thread
0  501  6460  6446  15   0   2504   772 -      R+   pts/2   0:00 ps lxH
```

Proces o identyfikatorze 6457 jest procesem wielowątkowym. Składają się na niego wątek główny i dwa wątki poboczne.

1.8.2 Zarządzanie wątkami

Wątek może oczekiwać na zakończenie innego wątku funkcją:

```
int pthread_join(pthread_t th, void **thread_return);
```

Jest to funkcja analogiczna do funkcji `wait()` oczekującej na zakończenie procesu potomnego. Wskaźnik `thread_return` zostanie zainicjowany wartością zwróconą przez wątek.

Wykonanie wątku można zakończyć w dowolnym momencie wywołaniem funkcji:

```
void pthread_exit(void *retval);
```

Wartość `retval` może być odczytana przez inny wątek, który zsynchronizuje się z nim wywołaniem `pthread_join()`.

Wątek można zatrzymać z poziomu innego wątku funkcją `pthread_cancel()`:

```
int pthread_cancel(pthread_t thread);
```

Zatrzymywanie wątku można blokować funkcją:

```
int pthread_setcancelstate(int state, int *oldstate);
```

Wątek może „odłączyć" się od procesu i kontynuować pracę niezależnie od wątku głównego. Służy do tego funkcja:

```
int pthread_detach(pthread_t th);
```

Uniezależnienie wątku oznacza, że jego zasoby będą zwolnione po jego zakończeniu. Z drugiej jednak strony nie będzie możliwe zsynchronizowanie z innym wątkiem poprzez wywołanie `pthread_join()`.

1.8.3 Obsługa sygnałów

Wszystkie wątki współdzielą między sobą jedną tablicę z adresami procedur obsługi. Każdy wątek może zmieniać przypisaną wcześniej do sygnału procedurę standardową funkcją `signal()`. Istnieje możliwość wysyłania sygnałów do poszczególnych wątków:

```
int pthread_kill(pthread_t thread, int signo);
```

Wątki mogą blokować odbiór określonych sygnałów używając funkcji:

```
int pthread_sigmask(int how, const sigset_t *newmask,
                              sigset_t *oldmask);
```

gdzie `newmask` jest maską sygnałów (4 liczby typu `unsigned long`) tworzoną z użyciem funkcji:

```
int sigemptyset(sigset_t *set);
int sigfillset(sigset_t *set);
int sigaddset(sigset_t *set, int signum);
int sigdelset(sigset_t *set, int signum);
int sigismember(const sigset_t *set, int signum);
```

Wywołanie funkcji `pthread_sigmask()` z argumentem drugim ustawionym na NULL powoduje odczytanie aktualnej maski sygnałów.

W ramach synchronizacji wątków można wymusić oczekiwanie na określony sygnał:

```
int sigwait(const sigset_t *set, int *sig);
```

1.8.4 Synchronizacja wątków

Do synchronizacji wątków wykorzystywane są następujące mechanizmy: zamki (ang. *mutex*), zmienne warunkowe (ang. *conditional variable*) i semafory POSIX.

1.8.4.1 Zamki

Zamki są binarnymi semaforami, a więc mogą znajdować się w jednym z dwóch stanów: podniesiony lub opuszczony. Do inicjowania zamka wykorzystywana jest funkcja:

```
int pthread_mutex_init(pthread_mutex_t *mutex,
                       const pthread_mutexattr_t *mutexattr);
```

Poniższy fragment kodu tworzy zamek z domyślnymi atrybutami:

```
pthread_mutex_t  m;
...
pthread_mutex_init(&m, NULL);
```

Do wykonywania operacji na zamkach służą następujące funkcje:

`pthread_mutex_lock()`
: Zajęcie zamka. Funkcja jest blokująca do czasu aż operacja może zostać wykonana.

`pthread_mutex_trylock()`
: Zajęcie wolnego zamka. Próba zajęcia już zajętego zamka kończy się zasygnalizowaniem błędu (`EBUSY`).

`pthread_mutex_unlock()`
: Zwolnienie zamka. Zwolnienia powinien dokonać wątek, który zajął zamek.

`pthread_mutex_destroy()`
: Usunięcie zamka.

Wszystkie wymienione funkcje pobierają wskaźnik do struktury `pthread_mutex_t` jako jedyny argument i zwracają wartość typu `int`. Przykład 1.14 pokazuje wykorzystanie zamków do implementacji prostego wzajemnego wykluczania.

1.8.4.2 Zmienne warunkowe

Zmienne warunkowe pozwalają na kontrolowane zasypianie i budzenie procesów w momencie zajścia określonego zdarzenia.

Poniższy fragment kodu tworzy zmienną warunkową z domyślnymi atrybutami:

```
pthread_cond_t  c;
...
pthread_cond_init(&c, NULL);
```

Budzenie wątków realizowane jest z wykorzystaniem jednej z dwóch poniższych funkcji:

```
int pthread_cond_signal(pthread_cond_t *cond);
int pthread_cond_broadcast(pthread_cond_t *cond);
```

Przykład 1.14: Proste zastosowanie zamków

```c
#include <stdio.h>
#include <unistd.h>
#include <pthread.h>

pthread_mutex_t mtx;

void* work(void* arg)
{
  int id = pthread_self();
  int i;
  for(i=0; i<10; i++)
  {
    printf("[%d] Czekam...\n", id);
    pthread_mutex_lock(&mtx);
    printf("[%d] Sekcja krytyczna...\n", id);
    sleep(1);
    printf("[%d] Wyjście...\n", id);
    pthread_mutex_unlock(&mtx);
    usleep(100000);
  }
  return NULL;
}

int main()
{
  pthread_t th1, th2;

  pthread_mutex_init(&mtx, NULL);
  pthread_create(&th1, NULL, work, NULL);
  pthread_create(&th2, NULL, work, NULL);
  pthread_join(th1, NULL);
  pthread_join(th2, NULL);

  return 0;
}
```

Funkcja `pthread_cond_signal()` budzi co najwyżej jeden wątek oczekujący na wskazanej zmiennej warunkowej. Funkcja `pthread_cond_broadcast()` budzi wszystkie wątki uśpione na wskazanej zmiennej warunkowej.

Kolejne dwie funkcje służą do usypiania wątku na zmiennej warunkowej:

```
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);
int pthread_cond_timedwait(pthread_cond_t *cond,
                           pthread_mutex_t  *mutex,
                           const struct timespec *abstime);
```

Funkcja `pthread_cond_wait()` oczekuje bezwarunkowo do czasu odebrania sygnału budzącego, podczas gdy funkcja `pthread_cond_timedwait()` ogranicza maksymalny czas oczekiwania. Zaśnięcie wątku wymaga użycia jednocześnie zmiennej warunkowej i zamka. Przed zaśnięciem zamek musi być już zajęty. Zaśnięcie oznacza atomowe zwolnienie zamka i rozpoczęcie oczekiwania na sygnał budzący. Obudzenie wątku powoduje ponowne zajęcie zamka. Poniższy przykład pokazuje zastosowanie zmiennej warunkowej do synchronizacji wątków:

- wątek oczekujący

```
pthread_cond_t  c;
pthread_mutex_t  m;
...
pthread_mutex_lock(&m);       /* zajęcie zamka */
pthread_cond_wait(&c, &m);    /* oczekiwanie na zmiennej warunkowej */
pthread_mutex_unlock(&m);     /* zwolnienie zamka */
```

- wątek budzący

```
pthread_cond_signal(&c);      /* sygnalizacja zmiennej warunkowej */
```

Pomimo, że nie jest to wymagane, często do poprawnej synchronizacji wywołuje się funkcję budzącą podczas przetrzymywania zamka:

```
pthread_mutex_lock(&m);
pthread_cond_signal(&c);
pthread_mutex_unlock(&m);
```

1.8.4.3 Semafory

Do synchronizacji wątków można wykorzystać semafory standardu POSIX. Jest to rozwiązanie całkowicie niezależne od semaforów Systemu V będących częścią zestawu mechanizmów komunikacji międzyprocesowej IPC. Semafor POSIX jest licznikiem przyjmującym wartości od zera wzwyż. Do obsługi semaforów POSIX przewidziano następujące funkcje:

`sem_init()`
: Funkcja tworząca i inicjująca nowy semafor. Semafor może być strukturą wewnętrzną procesu lub może służyć do synchronizacji niezależnych procesów, podobnie jak semafory IPC[6]. Argumentem funkcji `sem_init()` jest początkowa wartość semafora.

[6] Aktualna implementacja LinuxThreads nie wspiera semaforów synchronizujących niezależne procesy.

`sem_wait()`
: Oczekiwanie na wartość semafora większą od zera i obniżenie jej o 1.

`sem_trywait()`
: Nieblokująca próba zmniejszenia wartości semafora o 1.

`sem_post()`
: Zwiększenie wartości semafora o 1. Operacja zawsze wykonuje się poprawnie i nie jest blokująca.

`sem_getvalue()`
: Pobranie aktualnej wartości semafora.

`sem_destroy()`
: Usunięcie semafora.

Przykład 1.15 prezentuje proste zastosowanie semaforów standardu POSIX. Jest to oczywiście zmodyfikowana wersja przykładu 1.14.

Zadania

1. Przećwicz przekazywanie argumentów do wątku i odbiór wartości zwrotnych. Uruchom w tym celu 3 nowe wątki, które będą zwracały podwojoną wartość typu `int`.
2. Utwórz 2 wątki i wstrzymaj ich wykonanie zmienną warunkową. Wątek główny powinien wznowić ich pracę sygnalizując to odpowiedniej zmiennej warunkowej. Sprawdź różnicę pomiędzy funkcją `pthread_cond_signal()` a `pthread_cond_broadcast()`.
3. Zaproponuj implementację operacji `bariera()`, której działanie polegałoby na zsynchronizowaniu wskazanej liczby wątków. Operacja kończy się w momencie jej wywołania przez wszystkie wątki.
4. Zaproponuj synchronizację 2 wątków w problemie producenta-konsumenta. Przeanalizuj czy rozwiązanie to działa poprawnie również w przypadku większej liczby producentów i konsumentów.

Przykład 1.15: Proste zastosowanie semaforów POSIX

```c
#include <stdio.h>
#include <unistd.h>
#include <pthread.h>
#include <semaphore.h>

sem_t s;

void* work(void* arg)
{
  int id = pthread_self();
  int i;
  for(i=0; i<10; i++)
  {
    printf("[%d] Czekam...\n", id);
    sem_wait(&s);
    printf("[%d] Sekcja krytyczna...\n", id);
    sleep(1);
    printf("[%d] Wyjście...\n", id);
    sem_post(&s);
    usleep(100000);
  }
  return NULL;
}

int main()
{
  pthread_t th1, th2;

  sem_init(&s, 0, 1);
  pthread_create(&th1, NULL, work, NULL);
  pthread_create(&th2, NULL, work, NULL);
  pthread_join(th1, NULL);
  pthread_join(th2, NULL);

  return 0;
}
```

2
Java RMI

2.1 Podstawy

2.1.1 Wprowadzenie

Najkrócej ujmując, pakiet RMI (ang. *Remote Method Invocation*) to mechanizm obiektowego RPC dla języka Java. Pozwala on aplikacji klienta wywoływać metody zdalnych obiektów języka Java, czyli obiektów z innej przestrzeni adresowej, mogącej znajdować się na tej samej lub innej maszynie.

Celem niniejszego ćwiczenia jest zaprezentowanie podstaw tej technologii oraz konstrukcja nieskomplikowanej aplikacji rozproszonej z wykorzystaniem interfejsu RMI. W wykonaniu zadania pomoże prezentacja prostego programu, przedstawionego i objaśnionego w całości.

2.1.2 Procesy

Środowisko RMI obejmuje trzy procesy:

- proces *klienta*, wywołującego metody zdalnych obiektów,
- proces *serwera*, dostarczającego zdalnych obiektów,
- proces *rejestru* (ang. *object register*), wiążącego obiekty z ich nazwami. Obiekty są zgłaszane do rejestru przez serwer. Aplikacja klienta może uzyskać dostęp do zarejestrowanego obiektu pod warunkiem, że najpierw używając wcześniej poznanej nazwy obiektu, skorzysta z rejestru, by otrzymać referencję do obiektu odpowiadającą tej nazwie.

2.1.3 Etapy konstrukcji aplikacji

Podczas tworzenia trzech procesów, tworzących środowisko RMI, należy przestrzegać następujących zasad:

1. Interfejs obiektu i jego implementacja

   W pierwszej kolejności należy utworzyć interfejs zdalnego obiektu. Interfejs jest abstrakcją obiektu dostępną dla klienta. Rzeczywisty obiekt będący realizacją interfejsu znajduje się w serwerze i klient nie ma do niego dostępu. Interfejs zawiera

specyfikacje zdalnie dostępnych dla klienta metod. Każda z metod deklarowanych w interfejsie powinna zgłaszać wyjątek typu `java.rmi.RemoteException`.

Aby obiekt mógł być zdalnie dostępny, jego interfejs musi dziedziczyć z interfejsu `Remote`. Od klasy implementującej interfejs wymaga się zaś, by dziedziczyła z klasy `java.rmi.server.UnicastRemoteObject`.

2. Program serwera

   Serwer to proces, w którego przestrzeni adresowej znajduje się obiekt. Program serwera musi mieć dostęp zarówno do interfejsu, jak i do klasy go implementującej.

3. Program klienta

   Program klienta może być zrealizowany na różne sposoby, między innymi jako samodzielna aplikacja czy jako aplet języka Java. Od programu klienta wymaga się tylko, by miał dostęp do definicji interfejsu zdalnego obiektu.

4. Utworzenie pieńka (ang. *stub*) dla klienta. Pieniek powstaje na bazie skompilowanych klas odpowiadających interfejsowi i klasie go implementującej. Pieniek u klienta służy mu jako obiekt pomocniczy, reprezentujący faktyczny zdalny obiekt. W rzeczywistości klient odwołuje się bezpośrednio do swojego pieńka, zatem również referencja obiektu używana przez klienta jest w istocie referencją do jego pieńka. Dopiero w obiekcie pieńka następuje odwołanie do zdalnego obiektu. Pieniek zajmuje się również przekształcaniem wywołań metod na komunikaty protokołu sieciowego.

5. Aktywacja rejestru

   Model Java RMI wymaga istnienienia rejestru. Rejestr może zostać uruchomiony w procesie serwera (z wykorzystaniem klasy `LocateRegistry`) lub poza nim (polecenie `rmiregistry` w linii komend), może też działać na odrębnej maszynie.

2.1.4 Przykład prostej aplikacji

Poniżej przedstawiono i omówiono przykładową aplikację RMI. Jej działanie polega na wywołaniu w zdalnym obiekcie przez klienta metody, która wypisze po stronie serwera tekst na ekran.

2.1.4.1 Kod programu

Definicja interfejsu obiektu, zapisana w pliku o nazwie `HelloInterface.java` (zgodnie z wymogami języka java nazwa pliku musi być taka sama, jak nazwa interfejsu czy klasy), jest następująca:

```
import java.rmi.*;

public interface HelloInterface extends Remote
{
   void Hello(String message) throws RemoteException;
}
```

Interfejs zdalnego obiektu dziedziczy z interfejsu `Remote`. Metoda `Hello()` to jedyna operacja udostępniona klientom korzystającym z obiektu. A oto kod serwera, zapisany w pliku o nazwie `HelloServer.java`:

```
import java.net.*;
import java.rmi.*;
import java.rmi.server.*;
import java.rmi.registry.*;

public class HelloServer extends java.rmi.server.UnicastRemoteObject
                         implements HelloInterface
{
  Registry reg; // rejestr nazw obiektów

  /**
   * Metoda, implementująca funkcję Hello() interfejsu HelloInterface, którą
   * zdalnie wywołuje klient
   */
  public void Hello(String message) throws RemoteException
  {
    System.out.println(message);
  }

  public HelloServer() throws RemoteException
  {
    try
    {
      // Utworzenie rejestru nazw
      reg = LocateRegistry.createRegistry(1099);
      // związanie z obiektem nazwy
      reg.rebind("HelloServer", this);
    }
    catch(RemoteException e)
    {
      e.printStackTrace();
      throw e;
    }
  }

  public static void main(String args[])
  {
    // utworzenie obiektu dostępnego zdalnie
    try
    {
      HelloServer s = new HelloServer();
    }
    catch (Exception e)
    {
       e.printStackTrace();
       System.exit(1);
    }
  }
}
```

Klasa `HelloServer` jest zgodnie z wymogami pochodną klasy `UnicastRemoteObject`. Obiekt reg klasy `Registry` to rejestr nazw obiektów. Zostaje utworzony i uruchomiony wywołaniem `LocateRegistry.createRegistry` i nasłuchuje portu numer 1099 (domyślnego dla usługi rejestru). Jak zatem widać, przyjęto powyżej rozwiązanie, w którym rejestr utuchomiany jest w programie serwera. Wywołanie `reg.rebind("HelloServer", this)`

tworzy powiązanie obiektu (ściślej — referencji do obiektu) z nazwą, tutaj z nazwą "HelloServer". Klient, szukając obiektu w rejestrze, powinien posłużyć się właśnie tą nazwą. W funkcji `main()` w pierwszej kolejności następuje utworzenie zarządcy bezpieczeństwa, który umożliwi ewentualne ładowanie zdalnego kodu podczas przekazywania parametrów. Jego istnienie w przykładowym programie nie jest wymagane, ale staje się takim, gdy programy przesyłają parametry będące obiektami klasy `Serializable` (patrz następne ćwiczenie). W drugiej kolejności tworzony jest obiekt klasy `HelloServer`, implementującej zdalny interfejs. Po tej chwili obiekt przyjmuje i obsługuje wywołania klientów.

Poniżej zamieszczono kod programu klienta, zapisany oczywiście w pliku o nazwie `HelloClient.java`.

```
import java.net.*;
import java.rmi.*;
import java.rmi.registry.*;
public class HelloClient
{
  public static void main(String args[])
  {
    HelloInterface remoteObject;   // referencja do zdalnego obiektu
    Registry reg;                  // rejestr nazw obiektów
    String serverAddr=args[0];
    try
    {
      // pobranie referencji do rejestru nazw obiektów
      reg = LocateRegistry.getRegistry(serverAddr);
      // odszukanie zdalnego obiektu po jego nazwie
      remoteObject = (HelloInterface) reg.lookup("HelloServer");
      // wywołanie metody zdalnego obiektu
      remoteObject.Hello("Hello world!");
    }
    catch(RemoteException e)
    {
      e.printStackTrace();
    }
    catch(NotBoundException e)
    {
      e.printStackTrace();
    }
  }
}
```

Zmienna `remoteObject` jest referencją do zdalnego obiektu. Zmienna `reg` wskazuje na rejestr nazw obiektów. A w zmiennej `serverAddr` pamiętany jest (przekazany jako parametr wywołania programu) adres IP maszyny, na której działa serwer. Zakłada się tu, że serwer i rejestr działają na tej samej maszynie.

W pierwszej kolejności klient uzyskuje dostęp do rejestru nazw obiektów za pomocą wywołania `LocateRegistry.getRegistry(serverAddr)`. Następnie pobiera z rejestru referencję do obiektu o nazwie "HelloServer". Robi to, stosując funkcję `lookup` klasy `Register`:

```
remoteObject = (HelloInterface) reg.lookup("HelloServer");
```

Konieczne jest przy tym rzutowanie pozyskanej referencji na typ interfejsu zdalnego obiektu, tutaj `HelloInterface`. Dysponując wskazaniem na obiekt (ściślej — klient dys-

ponuje referencją do swojego pieńka, a dopiero ten ostatni — do zdalnego obiektu), klient wywołuje jego operacje tak, jakby obiekt był w jego lokalnej przestrzeni adresowej.

W powyższym prostym przykładzie klient nie przesyła nic do serwera. Oczywiście w typowej aplikacji takie przesłania, najczęściej w obie strony, mają miejsce. Przesyłanie wiadomości realizowane jest z reguły na dwa sposoby: poprzez przekazywanie w metodach parametrów predefiniowanych typów języka Java lub poprzez przekazywanie obiektów zdefiniowanych przez siebie klas. Bez względu na metodę, argumenty te stają się w rzeczywistości częścią komunikatu, przesyłanego przez sieć komputerową. W przypadku przesyłania obiektów zachodzi transfer kodu obiektu (ang. *code transfer* lub *code loading*) pomiędzy serwerem a klientem. Użycie tej metody jest przedmiotem innego ćwiczenia.

2.1.4.2 Kompilacja i uruchomienie programu

Podczas kompilacji i uruchomienia należy pamiętać o następujących zasadach:

- Program klienta powinien mieć dostęp do definicji interfejsu oraz do klasy implementującej interfejs
- Kod klasy implementującej interfejs powinien mieć dostęp do definicji interfejsu

Kompilacja programu serwera oraz klienta:

```
# javac HelloServer.java
# javac HelloClient.java
```

Utworzenie pieńka (ang. *stub*) dla klienta oraz szkieletu (ang. *skeleton*) dla serwera:

```
# rmic HelloServer
```

Polecenie to wygeneruje plik HelloServer_Stub.class, będący pieńkiem po stronie klienta.

Uruchomienie procesów:

```
# java HelloServer
# java HelloClient <adres IP serwera>
```

2.1.5 Zadanie

Należy zaprojektować i zrealizować z wykorzystaniem interfejsu Java RMI aplikację mini czat. Aplikacja powinna umożliwiać dołączanie nowych klientów do środowiska (ich liczba może być z góry ograniczona lub nawet stała), a następnie wymianę krótkich wiadomości pomiędzy nimi. To, co wyśle jeden klient, otrzymają następnie wszyscy pozostali. Przyjmowaniem wiadomości od klientów i powielaniem tych wiadomości powinien zajmować się zdalny obiekt, uruchomiony w serwerze. Dla ułatwienia, można się wzorować na przedstawionym wyżej programie.

2.2 Java RMI — przekazywanie parametrów

2.2.1 Wprowadzenie

W modelu Java RMI istnieją dwa sposoby przekazywania parametrów i zwracania wartości metod (dla uproszczenia, dalej w opracowaniu będzie mowa tylko o parametrach, choć zagadnienie dotyczy również wartości zwracanych przez metody): przez referencję

albo przez wartość. Przekazanie parametru przez referencję oznacza, że obiekt-parametr istnieje tylko w swojej pierwotnej lokalizacji, a odwołania do niego są zdalne, poprzez referencję do niego.

Przekazanie parametru przez wartość polega natomiast na *skopiowaniu* obiektu-parametru do maszyny wirtualnej, w której działa aplikacja wywołująca metodę zawierającą ten parametr. Późniejsze wywołania dokonywane są na tej kopii, czyli w lokalnej maszynie wirtualnej.

2.2.2 Przekazanie przez referencję

Przez referencję przekazywane są zawsze obiekty klas implementujących interfejs `Remote`. Ilustruje to poniższy przykład prostego interfejsu obiektu dokonującego inkrementacji (zwiększenie wartości o 1).

```
public interface IncrInterface extends Remote
{
  void Inc() throws RemoteException;
}
```

Ponieważ interfejs `IncInterface` dziedziczy z `java.rmi.Remote`, więc obiekty klasy `IncClass`, implementującej interfejs `IncInterface`, będą przekazywane przez referencję, i odwołanie do nich w rzeczywistości będzie zdalne:

```
public class IncClass extends java.rmi.server.UnicastRemoteObject
                      implements IncInterface
{
  int i;

  public void Inc() throws RemoteException
  {
    i++;
    System.out.println ("i = " + i);
  }

  public IncClass() throws RemoteException
  {
    i = 0;
  }
}
```

2.2.3 Przekazanie przez wartość

Poprzez wartość przekazywane są parametry niektórych klas języka Java oraz obiekty tych klas tworzonych przez użytkownika, które implementują interfejs `java.io.Serializable`. Rolą interfejsu `Serializable` jest umożliwienie przetworzenia dowolnego implementującego go obiektu do postaci umożliwiającej przesłanie go poprzez sieć komputerową (ang. *marshalling*). W odróżnieniu od klas typu `Remote`, klasom typu `Serializable` nie towarzyszy dodatkowa definicja interfejsu (gdyż nie są wywoływane zdalnie), przez co definiuje się je łatwiej niż klasy typu `Remote`.

Trzymając się powyższego przykładu obiektu dokonującego inkrementacji, definicja klasy, której obiekty będą przekazywane przez wartość, mogłaby wyglądać następująco:

```
public class IncClass implements java.io.Serializable
{
  int i;

  public void Inc() throws RemoteException
  {
    i++;
    System.out.println ("i = " + i);
  }

  public IncClass() throws RemoteException
  {
    i = 0;
  }
}
```

Jak widać, klasa `IncClass` nie implementuje żadnego dodatkowego interfejsu oprócz predefiniowanego `java.io.Serializable`.

2.2.4 Bezpieczeństwo

Począwszy od wersji 1.2 języka Java istnieje możliwość definiowania przez użytkownika zabezpieczeń maszyny wirtualnej Java. Zabezpieczenia te mogą obejmować definiowanie akceptowanych połączeń sieciowych (adresów IP, numerów portów), uprawnień do odczytu lub zapisu plików przez zewnętrzne aplikacje czy też uprawnień do ładowania kodu z innych maszyn wirtualnych Java. Ten ostatni przypadek szczególnie nas interesuje, gdyż przekazanie obiektu typu `Serializable` przez wartość powoduje załadowanie jego kodu i możliwość (a czasem — ryzyko) wykonywania jego metod.

W przypadku ładowania kodu z zewnątrz maszyna wirtualna Java wręcz *wymaga*, aby twórca aplikacji zdefiniował i wdrożył zasady bezpieczeństwa. Jeśli nie zostanie to zrobione, wykonanie programu zakończy następujący wyjątek:

```
java.security.AccessControlException: access denied \
(java.net.SocketPermission 127.0.0.1:1099 connect,resolve).
```

Definicja polityki bezpieczeństwa powinna się znaleźć w pliku o nazwie `java.policy`. Taki plik znajduje się zazwyczaj w podkatalogu `jre/lib/security` katalogu głównego instalacji środowiska Java, ale najczęściej nie jest używany do uruchamiania konkretnych aplikacji, gdyż zawiera tylko zbiór domyślnych, globalnych reguł. Zazwyczaj twórca aplikacji definiuje własny plik `java.policy`, którego następnie używa przy uruchamianiu aplikacji. Najprostsza możliwa zawartość pliku `java.policy`, nie zabezpieczająca maszyny wirtualnej w żaden sposób, jest następująca:

```
grant
{
  // Pelne uprawnienia dla wszystkich ladowanych zdalnie klas
  permission java.security.AllPermission;
};
```

Za odczytanie i realizowanie w programie Java zdefiniowanych wcześniej zasad bezpieczeństwa odpowiadają klasy tzw. *zarządców bezpieczeństwa*: ogólna — `SecurityManager` oraz specjalizowana (dla RMI) — `RMISecurityManager`. Zarządca bezpieczeństwa powinien zostać utworzony na samym początku funkcji `main()`, na przykład w następujący sposób:

```
public static void main(String args[])
{
  // utworzenie zarządcy bezpieczeństwa
  if (System.getSecurityManager() == null)
  {
    System.setSecurityManager(new RMISecurityManager());
  }
  ...
```

Wskazanie zarządcy bezpieczeństwa, gdzie powinien szukać definicji zasad bezpieczeństwa, następuje podczas uruchomienia programu. Przykładowo, komenda:

```
# java -Djava.security.policy=java.policy MyJavaProgram
```

wskazuje, że definicje zabezpieczeń (własność języka `java.security.policy`) znajdują się w pliku `java.policy`.

2.2.5 Zadanie

Należy zaprojektować i zaimplementować aplikację Java RMI, która zademonstruje różnice pomiędzy przekazywaniem parametrów przez referencję a przekazywaniem parametrów przez wartość. Obiekt przekazywany jako parametr powinien mieć przynajmniej jedną metodę wypisującą stan obiektu na ekran. Najistotniejsze jest zaobserwowanie, że

- stan obiektu przekazanego przez referencję (a więc dostępnego zdalnie) ulega ciągłym zmianom, gdyż obiekt znajduje się przez cały czas w jednej lokalizacji, w serwerze.
- obiekt przekazany przez wartość jest kopią, więc jego stan zostaje zainicjowany po stronie procesu, do którego trafia. Należy to zademonstrować.

Po której ze stron ukazują się komunikaty generowane przez metodę obiektu-parametru

a) W przypadku przekazania obiektu przez referencję?

b) W przypadku przekazania obiektu przez wartość?

Uwagi:

1. Nie jest istotne, jaki będzie tryb przekazania parametru (wejściowy czy wyjściowy). W szczególności, wartość zwracaną przez metodę również można traktować jako parametr — jest ona przykładem parametru wyjściowego.
2. W przypadku przkazania przez wartość parametrów typu `Serializable` należy uwzględnić informacje z punktu *Bezpieczeństwo*. Zaleca się, aby w obydwu programach (klienta i serwera) utworzyć obiekt zarządcy bezpieczeństwa.

3
CORBA

3.1 Wprowadzenie

CORBA (ang. *Common Object Request Broker Architecture*) to przemysłowy standard platformy programistycznej, służącej do budowania rozproszonych aplikacji obiektowych. Platformę CORBA wyróżnia to, że umożliwia współpracę aplikacji zaimplementowanych w różnych językach programowania, oraz istnienie bogatego zbioru usług pomocniczych ogólnego przeznaczenia (ang. *CORBA Services*).

3.2 Etapy tworzenia aplikacji

W pierwszej kolejności należy zaprojektować interfejs zdalnego obiektu. Jego definicję zapisuje się w specjalnym języku opisu interfejsu IDL (ang. *Interface Definition Language*). Interfejs ten jest następnie kompilowany do wybranego języka programowania. Program, który dokonuje tej kompilacji, nazywa się *komilatorem IDL*.

Wygenerowany na podstawie interfejsu kod objemuje przede wszystkim statycznego pośrednika IDL klienta (ang. *static IDL stub*) oraz szkielet (ang. *skeleton*) dla serwera. Zadaniem pośrednika IDL i szkieletu jest przetwarzanie komunikatów protokołu sieciowego na wywołania metod obiektu. Pośrednik IDL dodatkowo dostarcza aplikacji klienta pomocniczego obiektu (ang. *proxy*), którego interfejs jest taki sam, jak interfejs obiektu. Dzięki temu klient ma wrażenie, że pracuje bezpośrednio na zdalnym obiekcie, choć w rzeczywistości dokonuje wywołań na lokalnym obiekcie pomocniczym, który następnie przekazuje je za pośrednictwem ORB do zdalnego obiektu.

Najczęściej twórca obiektu generuje na podstawie definicji interfejsu również szablon *implementacji* obiektu, w którym następnie dopisuje kod metod obiektu.

Twórca obiektu oraz jego użytkownik muszą również dostarczyć programy: serwera, w którym obiekt zostanie utworzony, oraz klienta, który będzie wywoływał jego metody. Fragmenty kodu związane z utworzeniem obiektu CORBA są objęte standardem. Należy podkreślić, że programy klienta i serwera (czy obiektu) mogą zostać zaimplementowane w *różnych* językach programowania.

Ostatnim etapem przygotowania aplikacji CORBA jest kompilacja programów klienta i serwera do postaci wykonywalnej i ich uruchomienie.

3.3 Przykład prostej aplikacji

Niniejsze ćwiczenie ma na celu wprowadzenie platformy CORBA. Zostanie przedstawiona i omówiona prosta aplikacja, w której obiekt zdalny udostępnia jedną metodę, wypisującą na ekran maszyny serwera ciąg znaków przekazany jej przez klienta jako parametr. Programy klienta i serwera są zrealizowane w języku C++. Interfejs obiektu nazywa się `writer` i jest zapisany w pliku o nazwie `printer.idl`.

3.3.1 Interfejs obiektu

Interfejs obiektu `writer` jest zdefiniowany następująco (plik `writer.idl`):

```
interface writer
{
  void print (in string message);
};
```

Obiekt udostępnia poprzez swój interfejs jedną metodę o nazwie `print`. Przyjmuje ona wejściowy parametr typu `string` (jeden z predefiniowanych typów CORBA) o nazwie `message`; przekazuje go do obiektu klient, a zadaniem metody print będzie wyświetlenie jego wartości na ekranie maszyny serwera.

3.3.2 Kompilacja interfejsu

Kompilacji interfejsu IDL do wybranego języka programowania dokonuje się za pomocą tzw. *kompilatora IDL*. Dla każdego języka programowania wspieranego przez standard CORBA istnieje odrębny kompilator IDL (np. `idl` dla C++ czy `jidl` dla języka Java).

Kompilacja interfejsu `writer.idl`:

```
# idl --impl writer.idl
```

Opcja `--impl` oznacza żądanie dodatkowego wygenerowania szablonu implementacji obiektu (domyślnie nie jest on generowany). W wyniku kompilacji powstają następujące pliki:

- `writer.h` i `writer.cpp` — gotowy kod pośrednika IDL dla klienta
- `writer_skel.h` i `writer_skel.cpp` — gotowy kod szkieletu dla serwera
- `writer_impl.h` i `writer_impl.cpp` — szablon implementacji obiektu, który zostanie uzupełniony kodem

Nazwy generowanych (przyrostki `_impl` oraz `_skel`) plików są objęte standardem, aby możliwe było przenoszenie kodu pomiędzy różnymi implementacjami standardu.

3.3.3 Kod implementacji obiektu

W pliku `writer_impl.cpp` należy dopisać kod wewnątrz wygenerowanego szablonu metody *print* (dla uproszczenia poniżej ukazano tylko ten fragment, w którym następują modyfikacje):

```
1  //
2  // IDL:writer/print:1.0
3  //
4  void
5  writer_impl::print(const char* message)
6      throw(::CORBA::SystemException)
7  {
8      // TODO: Implementation
9
10    cout << message << endl;   // wypisanie ciągu znaków na ekran
11 }
```

Jak widać, w naszym prostym obiekcie implementacja metody sprowadza się do wypisania na ekran przekazanego w parametrze `message` ciągu znaków. Zastosowano tutaj operator strumieniowy „<<”, zdefiniowany w pliku nagłówkowym `iostream.h` (należy ten plik włączyć do programu).

3.3.4 Program serwera

W programie serwera w pierwszej kolejności należy uruchomić pośrednika ORB. Następnie trzeba utworzyć zdalnie dostępny obiekt i wskazać adapter, który ma zarządzać tym obiektem. Po utworzeniu obiektu można przyjmować i obsługiwać wywołania jego metod. Oto kod przykładowego serwera:

```
1  #include <OB/CORBA.h>
2  #include <writer_impl.h>
3  #include <fstream.h>
4  #include <iostream.h>
5
6  int main(int argc, char* argv[])
7  {
8     int status = EXIT_SUCCESS;
9     CORBA::ORB_var orb;
10    try
11    {
12       // inicjalizacja ORBa
13       orb = CORBA::ORB_init(argc, argv);
14
15       // utworzenie zarządcy adapterów POA
16       CORBA::Object_var poaObj = orb -> resolve_initial_references("RootPOA");
17       PortableServer::POA_var rootPoa = PortableServer::POA::_narrow(poaObj);
18
19      // utworzenie obiektu CORBA
20       writer_impl* obj = new writer_impl (rootPoa);
21       writer_var writerObj = obj -> _this();
22
23       // zapisanie odniesienia do obiektu w pliku
24       ofstream out("IOR.ref");
25       out << orb -> object_to_string(writerObj) << endl;
26       out.close();
27
28       // aktywacja adaptera i ORB
29       rootPoa -> the_POAManager() -> activate();
30       orb -> run();
31    }
```

```
32      catch (const CORBA::Exception&)
33      {
34         status = EXIT_FAILURE;
35      }
36      if (!CORBA::is_nil(orb))
37      {
38         try
39         {
40            orb -> destroy();
41         }
42         catch (const CORBA::Exception&)
43         {
44            status = EXIT_FAILURE;
45         }
46      }
47      return status;
48   }
```

Powyższy program jest wprawdzie stosunkowo długi, jak na prostą aplikację, ale należy pamiętać, że jego kod jest w zasadzie standardowy — proces inicjalizacji obiektu przebiega zawsze tak samo, a zmianie podlega tylko nazwa jego klasy oraz nazwy zmiennych użytych w programie. Szablony kodu serwera i klienta są — jako standardowe — publicznie dostępne, na przykład w specyfikacjach CORBA. Poniżej omówiono najważniejsze fragmenty kodu serwera.

Przede wszystkim, zaimportowany został plik nagłówkowy `writer_impl.h`, zawierający definicję klasy implementującej zdalny obiekt oraz klasy realizującej szkielet. Na samym początku funkcji `main()` następuje inicjalizacja pośrednika ORB (operacja `ORB_init()`). Zaraz po tym pośrednik pobiera wskazanie do domyślnego adaptera obiektów (wywołanie `resolve_initial_references("RootPOA")`; adapter taki posiada domyślne polityki aktywacji). W kolejnej linii następuje zawężenie obiektu adaptera, będącego póki co ogólnym obiektem `CORBA::Object_var`, do typu adaptera `PortableServer::POA_var`.

Zdalny obiekt `writer` w serwerze to w rzeczywistiści obiekt klasy implementującej go, w tym przypadku klasy `writer_impl`. Tworząc go, jako parametr konstruktora należy przekazać adapter, który ma obsługiwać dany obiekt. W kolejnej linii utworzone zostaje odniesienie do obiektu CORBA (wywołanie `_this()`). W następnych trzech liniach następuje przekształcenie odniesienia do obiektu na postać ciągu znaków (wywołanie `object_to_string` pośrednika ORB) i zapisanie wyniku w pliku `IOR.ref`. Plik ten należy następnie w jakiś sposób udostępnić programowi klienta, by mógł on na podstawie zawartego w pliku odniesienia zlokalizować zdalny obiekt.

Uwaga — odniesienie w postaci ciągu znaków jest *niezależne od języka programowania*. Ma taką samą postać bez względu na to, w jakim języku napisano tworzący je serwer.

3.3.5 Program klienta

Podobnie jak dla serwera, również po stronie klienta istnieje konieczność uruchomienia pośrednika ORB. Posłuży on następnie klientowi do odnalezienia zdalnego obiektu i wywoływania jego metod. I znów, w zasadzie cały kod klienta (z wyjątkiem fragmentów związanych z wywoływaniem zdalnego obiektu) jest standardowy. Oto program klienta:

```
1   #include <OB/CORBA.h>
2   #include <writer.h>
```

```
3  #include <fstream.h>
4  #include <iostream.h>
5  #include <unistd.h>
6
7  int main(int argc, char* argv[])
8  {
9     int status = EXIT_SUCCESS;
10    CORBA::ORB_var orb;
11    try
12    {
13       // iniclalizacja ORBa
14       orb = CORBA::ORB_init(argc, argv);
15       // utworzenie referencji do obiektu CORBA
16       CORBA::Object_var obj = orb -> string_to_object("relfile:/IOR.ref");
17       writer_var writerObj = writer::_narrow(obj);
18       writerObj->print("Hello, server !!!");
19    }
20    catch (const CORBA::Exception&)
21    {
22       status = EXIT_FAILURE;
23    }
24    if (!CORBA::is_nil(orb))
25    {
26       try
27       {
28          orb -> destroy();
29       }
30       catch (const CORBA::Exception&)
31       {
32          status = EXIT_FAILURE;
33       }
34    }
35    return status;
36 }
```

Po uruchomieniu pośrednika ORB klienta następuje utworzenie odniesienia do obiektu pomocniczego (ang. *proxy*), na którym klient będzie bezpośrednio wywoływał operacje zdalnego obiektu. Odniesienie to jest pozyskiwane z pliku (za pomocą operacji `string_to_object`), który wcześniej został udostępniony twórcy programu klienta. W dalszej kolejności następuje zawężenie obiektu do typu `writer`, a po nim — wywołanie metody.

3.3.6 Kompilacja i uruchomienie programów

Dla poniższej części ćwiczenia zakłada się wykorzystanie implementacji standardu CORBA firmy IONA Technologies, znanej pod nazwą ORBacus, oraz systemu operacyjnego Linux. Pliki nagłówkowe zapisane są domyślnie w katalogu `/usr/local/include`, a skompilowane biblioteki - w katalogu `/usr/local/lib`. W przypadku bibliotek dynamicznych należy pamiętać o ustawieniu w systemie Linux zmiennej systemowej `LD_LIBRARY_PATH` tak, by wskazywała katalog zawierając skompilowane biblioteki (tutaj `/usr/local/lib`). Potrzebne są cztery biblioteki: dwie systemowe, `libdl` i `libpthread`, oraz dwie dla wywołań CORBA, `libOB` (implementacja pośrednika ORB) i `libJTC` (emulacja wątków języka Java w C++). Poniżej przedstawione zostaną kolejno wszystkie polecenia kompilacji, można jednak skrócić proces kompilacji przez zapisanie tych poleceń w skrypcie lub przez

użycie narzędzia make.

```
# gcc -c -I. -I/usr/local/include writer.cpp
# gcc -c -I. -I/usr/local/include writer_skel.cpp
# gcc -c -I. -I/usr/local/include writer_impl.cpp
# gcc -c -I. -I/usr/local/include client.cpp
# gcc -c -I. -I/usr/local/include server.cpp
# gcc -o client client.o writer.o -L/usr/local/lib -lOB -ldl -lJTC -lpthread
# gcc -o server server.o writer.o writer_skel.o writer_impl.o \
      -L/usr/local/lib -lOB -ldl -lJTC -lpthread
```

Uruchomienie programów serwera i klienta:

```
# ./server
# ./client
```

Wynik w programie serwera:

```
Hello, server !!!
```

Zadania

1. Zaimplementować i uruchomić aplikację writer. Do wykonania ćwiczenia pomocny może się okazać przedstawiony powyżej kod.
2. Zapoznać się z użyciem typu any, uogólniającego wszystkie (również obiektowe) typy CORBA. Użyć w wywołaniu metody print argumentu tego typu zamiast typu string, a następnie ponownie zaimplementować tak zmodyfikowany obiekt.
3. Zaimplementować i uruchomić zmodyfikowaną aplikację writer, w której strony klienta i serwera zostaną zrealizowane *w różnych językach programowania*, C++ i Java (w dowolnej kombinacji). W przypadku języka Java pomocny będzie podręcznik [ION01].

4 Network File System

Network File System to rozproszony system plików stworzony przez firmę Sun Microsystems.

4.1 Konfiguracja klienta

Zanim klient będzie w stanie odwołać się do zasobów zdalnego systemu, musi poznać listę udostępnianych zasobów. Może to wykonać za pomocą komendy `showmount(8)`:

```
# showmount -e localhost
mount clntudp_create: RPC: Program not registered
# showmount -e unixlab
Export list for unixlab:
/home      lab1,lab2,lab3
```

Przełącznik `-e` (od ang. *export*) pobiera listę zasobów (katalogów) ze wskazanego komputera czyli katalogów *eksportowanych* przez ten system. W powyższym przypadku komputer klienta nie udostępnia żadnego katalogu, bo nie został uruchomiony jeszcze serwer NFS. Implementacja serwera NFS jest oparta na mechanizmie RPC, a komunikat pochodzi od usługi `portmap` wskazując, że nie ma zarejestrowanej wymaganej usługi RPC. Komputer `unixlab` udostępnia katalog `/home` i udostępnia go komputerom `lab1`, `lab2`, `lab3`.

Klient systemu NFS jest wbudowany w jądro systemu operacyjnego. Uzyskanie dostępu do zdalnego katalogu wymaga dołączenia zdalnego systemu plików analogicznie do tego, jak wykonywane jest to lokalnie. Montowanie katalogu wymaga użycia komendy `mount(8)`:

```
mount system_plików katalog_docelowy
```

Przykładem zastosowania komendy `mount` jest dołączenie systemu plików z dyskietki lub napędu CD:

```
# mount /dev/fd0 /mnt
```

W przypadku dysków CD należy użyć nazwy `/dev/cdrom`. Katalog `/mnt` bardzo często wykorzystywany jest do tymczasowego dołączania systemów plików. Po poprawnym wykonaniu komendy `mount` zawartość katalogu `/mnt` powinna być tożsama z zawartością

katalogu `/home` na serwerze `unixlab`. Przetwarzanie plików w katalogu `/mnt` może być realizowane w taki sam sposób jak innych plików lokalnych.

Odłączenie systemu plików, zarówno lokalnego jak i zdalnego, wykonywane jest komendą `umount(8)`:

```
# umount /mnt
```

Parametrem komendy może być zarówno nazwa katalogu jak i nazwa systemu plików. Katalog musi być niewykorzystywany, tzn. żaden proces nie może mieć otwartego pliku z podkatalogów tego katalogu i żaden proces nie może mieć skojarzonego żadnego z podkatalogów jako swojego katalogu bieżącego.

Dołączając system plików można wskazać jakiego jest on typu. System Linux obsługuje m.in. następujące systemy plików:

`ext2`	standardowy system plików systemu Linux
`ext3`	rozszerzona o księgowanie wersja systemu `ext2`
`vfat`	system plików FAT z obsługą długich nazw plików
`ntfs`	system plików NTFS z Windows NT/200x/XP
`iso9660`	system plików z dysków CD
`udf`	system plików UDF (np. DVD-ROM)
`nfs`	sieciowy system plików NFS
`smbfs`	system plików SMB (zobacz opis pakietu Samba)
`reiserfs`	system plików ReiserFS
`xfs`	system plików XFS

Dołączenie systemu plików z dyskietki formatowanej w systemie Windows można więc wykonać następująco:

```
# mount -t vfat /dev/fd0 /mnt
```

W przypadku systemu NFS identyfikacja systemu plików wymaga wskazania zdalnego serwera i katalogu na tym serwerze. Dołączenie katalogu `/home` z systemu `unixlab` można więc zrealizować następująco:

```
# mount -t nfs unixlab:/home /mnt
```

Dołączanie systemów plików może być parametryzowane dodatkowymi opcjami definiowanymi przełącznikiem `-o`. Oto krótka lista ważniejszych opcji montowania:

`ro`	tryb tylko do odczytu
`atime`	włącznie aktualizacji daty ostatniego dostępu
`dev`	włączenie interpretacji plików specjalnych reprezentujących urządzenia
`exec`	zezwolenie na wykonywanie programów
`suid`	włączenie interpretacji bitów SUID i SGID
`sync`	synchroniczne wykonywanie operacji dostępu
`async`	asynchroniczne wykonywanie operacji dostępu
`remount`	zmiana parametrów montowania

Niektóre opcje występują również w wersjach negujących ich działanie z dodatkowym prefiksem `no`, np. `noatime`, `nodev`, `noexec`, `nosuid`. Dołączenie zdalnego systemu plików w trybie tylko do odczytu można więc wykonać następująco:

```
# mount -o ro unixlab:/home /mnt
```

W przykładzie pominięto specyfikację typu systemu plików, gdyż wynika ona bezpośrednio z formatu reprezentacji urządzenia (*nazwa_komputera*`:`*katalog*).

System plików NFS obsługuje wiele dodatkowych opcji montowania systemu plików. Pełną listę można znaleźć na stronie pomocy systemowej `nfs(5)`. Oto ważniejsze z nich:

`hard`	nieprzerwane odwoływanie się do serwera w przypadku awarii
`soft`	sygnalizacja błędu na poziomie aplikacji w przypadku awarii serwera
`intr`	umożliwia przerwanie operacji wejścia/wyjścia poprzez sygnał w przypadku montowania miękkiego (opcja `soft`)
`tcp`	połączenie z serwerem za pośrednictwem protokołu TCP
`rsize=n`	rozmiar bufora odczytu z serwera NFS
`wsize=n`	rozmiar bufora zapisu z serwera NFS
`timeo=n`	czas oczekiwania na odpowiedź przed retransmisją żądania
`retrans=n`	maksymalna liczba retransmisji przed zgłoszeniem błędu w trybie `soft`
`acreqmin=n`	minimalny czas przechowywania informacji o plikach zwykłych w pamięci podręcznej
`acreqmax=n`	maksymalny czas przechowywania informacji o plikach zwykłych w pamięci podręcznej
`acdirmin=n`	minimalny czas przechowywania informacji o katalogach w pamięci podręcznej
`acdirmax=n`	maksymalny czas przechowywania informacji o katalogach w pamięci podręcznej
`noac`	całkowite wyłączenie pamięci podręcznej dla atrybutów plików

4.2 Konfiguracja serwera

Konfiguracja serwera jest bardzo prosta i wymaga modyfikacji pliku konfiguracyjnego `/etc/exports(5)`. Plik ten zawiera listę katalogów, które będą udostępniane innym systemom. Oto przykład takiego pliku:

```
/home   lab1(rw)
```

Po nazwie katalogu znajduje się lista komputerów, które będą miały prawo odwoływania się do tego katalogu. W nawiasach znajdują się dodatkowe opcje eksportowania takiego katalogu. Lista komputerów może zawierać nazwy w postaci domenowej (również z użyciem znaków uogólniających), adresy IP komputerów (i adresy sieci), odwołania do grup sieciowych. Oto przykład bardziej rozbudowanej konfiguracji:

```
/home   *.cs.put.poznan.pl(ro)  lab?.cs.put.poznan.pl(rw)
/usr    192.168.0.0/255.255.255.0(async,rw)
/export @lab
```

W przykładzie katalog /home został udostępniony wszystkim komputerom z domeny cs.put.poznan.pl do odczytu i wszystkim komputerom o nazwach zaczynających się od lab z tej samej domeny w trybie do zapisu. Katalog /usr jest udostępniany wszystkim komputerom o adresach od 192.168.0.1 do 192.168.0.254 w trybie asynchronicznym. Katalog /export udostępniany jest wszystkim komputerom z grupy sieciowej lab (zobacz dalej). Szczegółowy opis dostępnych opcji można znaleźć na stronie pomocy systemowej exports(5).

4.2.1 Uruchomienie serwera

Uruchomienie serwera wymaga wystartowania odpowiednich procesów usługowych. System NFS implementowany jest poprzez 2 protokoły (mountd i nfs), stąd 2 programy usługowe. Po uruchomieniu pierwszego z nich:

```
# rpc.mountd
```

będzie możliwe odpytanie serwera o listę dostępnych zasobów:

```
# showmount -e
Export list for localhost:
/export @lab
/home   lab?.cs.put.poznan.pl,*.cs.put.poznan.pl
/usr    192.168.0.0/255.255.255.0
```

Próba dołączenia systemu plików skończy się jednak zawieszeniem operacji mount. Do poprawnej pracy niezbędny jest drugi proces usługowy:

```
# rpc.nfsd
```

odpowiedzialny za właściwy dostęp do zawartości poszczególnych plików.

4.2.2 Rekonfiguracja

Każda zmiana wprowadzona do pliku /etc/exports wymaga powiadomienia procesów usługowych serwera NFS. Można to zrobić wysyłając sygnał HUP do *obu* procesów usługowych:

```
# ps -ax
...
# kill -HUP 2345  2456
```

Wartości 2345 i 2456 są identyfikatorami procesów rpc.mountd i rpc.nfsd. Można również wykorzystać do tego celu polecenia pkill(1):

```
# pkill -HUP rpc.mountd
# pkill -HUP rpc.nfsd
```

lub killall(1):

```
# killall -HUP rpc.mountd rpc.nfsd
```

Niektóre implementacje serwera NFS udostępniają komendę exportfs, która m.in. umożliwia aktualizację konfiguracji serwera:

```
# exportfs -r
```

Powyższe polecenie umożliwia również dynamiczne dodawanie i usuwanie katalogów przeznaczonych do udostępnienia, co ilustruje poniższy przykład:

```
# exportfs        <== działa analogicznie do showmount -e
/usr              192.168.0.0/255.255.255.0
/home             *.cs.put.poznan.pl
/home             lab?.cs.put.poznan.pl
# exportfs -u '*.cs.put.poznan.pl:/home'
# exportfs -v
/usr              192.168.0.0/255.255.255.0(rw,async,wdelay,root_squash)
# exportfs -o ro lab1.cs.put.poznan.pl:/home
# exportfs -v
/usr              192.168.0.0/255.255.255.0(rw,async,wdelay,root_squash)
/home             lab1.cs.put.poznan.pl(ro,async,wdelay,root_squash)
```

Przełącznik `-u` umożliwia usuwanie katalogu z listy udostępnianych katalogów. Przełącznik `-v` włącza wyświetlanie dodatkowych informacji (np. opcji eksportowania katalogu). Przełącznik `-o` powoduje ustawienie opcji eksportowanego katalogu.

4.2.3 Grupy sieciowe

Ograniczanie dostępu do katalogów na serwerze NFS może się odbywać w oparciu o grupy sieciowe. Grupa sieciowa konfigurowana jest w pliku `/etc/netgroup(5)` i składa się z trójek wartości:

(*komputer*, *użytkownik*, *domena*)

Każda z tych wartości może być pominięta. Z punktu widzenia zastosowania grup sieciowych do konfiguracji systemu NFS najważniejsza jest możliwość definiowania grup komputerów. Poniższy przykład pokazuje definicję grupy `lab` składającej się z komputerów `lab1`, `lab2`, `lab3`:

```
lab   (lab1, , )  (lab2, , )  (lab3, , )
```

W pliku konfiguracyjnym `/etc/exports` odwołania do grup sieciowych poprzedzone są znakiem `@`:

```
/home   @lab(rw,async,no_root_squash)
```

4.3 Mapowanie użytkowników

Identyfikacja użytkowników w systemie NFS odbywa się wg. ich identyfikatorów numerycznych. Oznacza to, że w przypadku niezgodności w konfiguracji użytkowników pomiędzy systemami klienta i serwera może dojść do nieuprawnionego dostępu do danych. Z tego powodu w domyślnej konfiguracji konto użytkownika `root` podlega mapowaniu na użytkownika o identyfikatorze -2 (65534). Działanie to można kontrolować następującymi parametrami eksportowania katalogów:

`root_squash`
: włączenie mapowania użytkownika o identyfikatorze 0 (`root`) na użytkownika o identyfikatorze -2 (`nobody`)

`no_root_squash`
: wyłączenie mapowania użytkownika 0

`all_squash`
: mapowanie wszystkich użytkowników do użytkownika o identyfikatorze -2

`anonuid`
: identyfikator użytkownika anonimowego

`anonguid`
: identyfikator grupy użytkownika anonimowego

Poniższy przykład pokazuje konfigurację serwera NFS udostępniającego katalogi domowe użytkownikom systemów innych niż Unix:

```
/home/pc1   pc1(rw,all_squash,anonuid=1001,anongid=100)
/home/pc2   pc2(rw,all_squash,anonuid=1002,anongid=100)
/home/pc3   pc3(rw,all_squash,anonuid=1003,anongid=100)
```

W przykładzie użytkownik użytkownik o identyfikatorze 1001 korzysta z komputera o nazwie `pc1` i należy do grupy o identyfikatorze 100. Serwer NFS udostępnia jego katalog domowy tylko dla komputera `pc1`. Analogicznie użytkownicy 1002 i 1003 korzystają z komputerów `pc2` i `pc3`.

4.3.1 Mapowanie statyczne

Serwer może wprowadzić statyczną tabelę odwzorowań użytkowników z systemu klienta na użytkowników serwera. Wymaga to wyspecyfikowania dodatkowej opcji w konfiguracji wskazującej na plik z odwzorowaniami:

```
/home   *.cs.put.poznan.pl(rw,map_static=/etc/nfs/home.map)
```

Zawartość pliku `/etc/nfs/home.map` definiuje odwzorowania poszczególnych użytkowników:

```
uid  0-99      -      # mapowanie do nobody
uid  100-500   1000   # mapowanie 100-500 na 1000-1400
gid  0-49      -      # mapowanie id grup 0-49 na -2
gid  50-100    700    # mapowanie grup 50-100 na 700-750
```

4.3.2 Mapowanie dynamiczne

Wadą mapowania statycznego jest oczywiście jego statyczność. Rozwiązanie dynamiczne polega na uruchomieniu po stronie *klienta* dodatkowej usługi, której zadaniem jest udostępnianie informacji o odwzorowaniach identyfikatorów użytkowników na nazwy. W celu należy po stronie klienta uruchomić proces:

```
# rpc.ugidd
```

a do konfiguracji serwera dopisać nową opcję:

```
/home   *.cs.put.poznan.pl(rw,map_daemon)
```

4.4 WebNFS

WebNFS to rozszerzenie standardowego protokołu NFS o możliwość pobierania plików za pośrednictwem publicznego uchwytu do plików. Umożliwi to proste pobieranie plików np. przez przeglądarki internetowe. Konfiguracja wymaga wskazania głównego katalogu dla WebNFS i udostępnienia wybranych katalogów w zwykły sposób. Oto przykładowa konfiguracja:

```
/test =public
/test *.cs.put.poznan.pl(ro,all_squash,insecure)
```

W przeglądarce należy wpisać odpowiedni adres URL:

```
nfs://galio.cs.put.poznan.pl/plik.txt
```

Przeglądarka Konqueror obsługuje rozszerzenie WebNFS.

4.5 Automonter

Dołączanie systemów plików może być automatyzowane za pomocą uzupełniającego oprogramowania uruchamianego po stronie klienta o nazwie *automounter*. Konfiguracja automontera opiera się na głównym pliku konfiguracyjnym `/etc/auto.master(5)` oraz na plikach uzupełniających `/etc/auto.*`. W pliku `/etc/auto.master` wskazywane są katalogi, które będą kontrolowane przez oprogramowanie automontera:

```
/home    /etc/auto.home
/a       /etc/auto.a
```

Nazwa pliku pojawiająca się po nazwie katalogu wskazuje na szczegółowy plik konfiguracyjny opisujący dany katalog. Skrypt startowy automontera dla każdego katalogu wymienionego w pliku `/etc/auto.master` uruchamia komendę `automount` wskazując plik konfiguracyjny. W przypadku katalogu `/home` zostanie więc wydana komenda:

```
# automount /home file /etc/auto.home
```

Argument `file` wskazuje, że konfiguracja znajduje się w zwykłym pliku o nazwie wskazanej trzecim argumentem. Dane dla automontera mogą również pochodzić z systemu NIS (zobacz punkt 6) lub mogą być generowane dynamicznie przez program. Szczegółowy opis konfiguracji znajduje się stronie pomocy systemowej `autofs(5)`.

Działanie automontera sprowadza się do monitorowania odwołań do podkatalogów kontrolowanego katalogu. W przypadku stwierdzenia odwołania do zdefiniowanego katalogu następuje jego dołączenie, zgodnie ze wskazaniami z pliku konfiguracyjnego. Poniższy przykład prezentuje konfigurację z pliku `/etc/auto.home`:

```
user1                   unixlab:/home/user1
user2                   galio:/export/home/user2
user3  -fstype=ext3,rw  :/dev/sda4
```

W pierwszej kolumnie znajdują się nazwy podkatalogów konfigurowanego katalogu. Druga kolumna (opcjonalna) zawiera dodatkowe opcje montowania systemu plików. Trzecia kolumna to wskazanie na system plików. Zakładając powyższą konfigurację wykonanie komendy:

```
# ls /home/user1
```

spowoduje dołączenie katalogu zdalnego /home/user1 z systemu unixlab. Katalog domowy użytkownika user3 dołączany jest z lokalnego systemu plików, co powoduje wykonanie przez automontera komendy:

```
# mount -o rw -t ext3 /dev/sda4 /home/user3
```

W przypadku konfiguracji katalogów domowych użytkowników, których pliki znajdują się w podkatalogach jednego wspólnego katalogu, można zastosować zapis skrócony:

```
*       unixlab:/export/home/&
```

Znak * oznacza w tym miejscu dowolny ciąg znaków, który zostanie następnie użyty w miejscu wystąpienia znaku &. Odwołanie do katalogu /home/abc spowoduje więc próbę dołączania katalogu /export/home/abc z systemu unixlab.

4.5.1 Zwiększanie dostępności

Automonter umożliwia zwiększanie dostępności zdalnych katalogów poprzez ich replikację. Zakładając, że serwery srv1 i srv2 udostępniają to samo oprogramowanie w katalogu /usr/local, można rozważyć następującą konfigurację:

```
local       / srv2:/usr/local / srv1:/usr/local
```

W przykładzie wskazano na dwa źródła dla podkatalogu local co oznacza, że niemożność dołączenia tego katalogu z systemu srv1 spowoduje dołączenie go z srv2. Ponieważ cała operacja będzie niewidoczna dla użytkownika, jedynym efektem tej konfiguracji będzie zwiększona niezawodność zdalnego katalogu. Parametr poprzedzający wskazanie na dalszy serwer oznacza dodatkowy podkatalog katalogu local, który pojawi się w momencie dołączenia systemu plików, co może umożliwić rozróżnienie przez użytkownika źródła danych.

4.5.2 Dynamiczna konfiguracja

Dane dla automontera mogą pochodzić z programu dynamicznie generującego konfigurację. Wymaga to uruchomienia programu automount z opcją program:

```
# automount /a program /etc/auto.sh
```

W powyższym przykładzie konfigurację dostarcza skrypt /etc/auto.sh. Program ten jest wywoływany z argumentem będącym nazwą podkatalogu, do którego odwołuje się użytkownik. Oto przykładowa implementacja w języku sh:

```
#!/bin/sh

echo "unixlab.cs.put.poznan.pl:/export/home/$1"
```

Skrypt umożliwia dostęp do katalogów domowych wszystkich użytkowników. Specjalnym zastosowaniem programów generujących zapisy konfiguracyjne jest równoważenie obciążenia serwerów. Zakładając konfigurację z poprzedniego punktu (podkatalog local), program konfiguracyjny mógłby zwracać naprzemiennie wskazania na serwery srv1 i srv2.

Dokumentacja

Strony pomocy systemowej: `mount(8)`, `nfs(5)`, `autofs(5)`, `autofs(8)`, `automount(8)`, `auto.master(5)`.

Zadania

1. Dołącz wybrany system plików z dyskietki lub CD-ROMu do katalogu `/mnt`.
2. Dołącz wybrany system plików NFS do katalogu `/mnt`.
3. Skonfiguruj serwer NFS udostępniając przykładowy katalog, np. `/test`. Udostępnij katalog w trybie do odczytu, a następnie do zapisu dla wybranych komputerów. Zweryfikuj możliwość zapisu w katalogu zdalnym z poziomu użytkownika `root` i zwykłego użytkownika.
4. Sprawdź odporność klienta systemu NFS na przejściowe awarie serwera. W tym celu zatrzymaj na chwilę działanie procesów serwera NFS.
5. Sprawdź różnicę między trybami pracy dołączania zdalnego katalogu: `hard`, `soft` i połączenie `hard,intr`.
6. Sprawdź działanie pamięci podręcznej po stronie klienta NFS. W tym celu jednocześnie wyświetlaj zawartość katalogu po stronie klienta i zmieniaj ją po stronie serwera. Ćwiczenie wykonaj również po całkowitym wyłączeniu mechanizmu pamięci podręcznej.
7. Udostępnij katalog do zapisu dla wszystkich użytkowników łącznie z użytkownikiem `root`.
8. Skonfiguruj serwer NFS do pracy w trybie statycznego mapowania użytkownika ze stacji roboczej o identyfikatorze 1500 na użytkownika o tej samej nazwie i identyfikatorze 1600 po stronie serwera. Zastosuj mapowanie statyczne i dynamiczne.
9. Udostępnij katalog w trybie WebNFS.
10. Skonfiguruj oprogramowanie automontera tak, aby umożliwiało dołączanie systemów plików z napędów CD-ROM i zdalnego katalogu poprzez system plików NFS.
11. Sprawdź możliwość zwiększania niezawodności dostępu do replikowanych danych poprzez odpowiednią konfigurację automontera.

5
Pakiet Samba

5.1 Wprowadzenie

Pakiet Samba jest oprogramowaniem umożliwiającym integrację systemów Unix i Windows. Samba implementuje protokół SMB wykorzystywany m.in. w sieciach Microsoft Network. Poprawne skonfigurowanie pakietu umożliwia:

- udostępnianie katalogów i drukarek z systemu Unix,
- dostęp w systemie Unix do katalogów i drukarek udostępnionych przez inne systemy,
- wykorzystanie protokołu komunikacyjnego WinPopup w systemach Unix,
- zarządzanie domeną Windows NT z poziomu systemu Unix,
- autoryzacja użytkowników za pośrednictwem istniejącego kontrolera domeny NT.

Na Sambę składają się m.in. następujące programy:

`smbd` demon udostępniający lokalne pliki i drukarki. Konfiguracja demona zapisana jest w pliku `smb.conf`.

`nmbd` demon obsługujący protokół NetBIOS — usługę serwisu nazw, umożliwiającą również przeglądanie list dostępnych zasobów. `nmbd` może pracować jako serwer WINS (Windows Internet Name Server).

`smbclient` dostarcza prostego interfejsu dostępu do współdzielonych zasobów. Działa na zasadzie programu FTP. Umożliwia dostęp do zdalnych plików i drukarek.

`testparm` narzędzie pomocnicze wspomagające administratora w konfigurowaniu demona `smbd`. Sprawdza poprawność pliku konfiguracyjnego `smb.conf`.

`smbstatus` umożliwia śledzenie bieżących połączeń lokalnego serwera.

Pakiet Samba rozprowadzany jest na zasadach licencji GNU, a więc jest darmowym i publicznie dostępnym oprogramowaniem.

5.2 Klient protokołu SMB

Dostęp do zdalnych zasobów udostępnianych za pośrednictwem protokołu SMB wymaga wstępnego pobrania informacji o tych zasobach. Pierwszym krokiem do tego celu jest przeszukanie lokalnej podsieci w poszukiwaniu dostępnych serwerów:

```
# findsmb
IP ADDR         NETBIOS NAME    WORKGROUP/OS/VERSION
---------------------------------------------------------------------
192.168.0.45    GALIO            [TEST] [Windows 5.0] [Windows 2000]
192.168.0.1     UNIXLAB         +[TEST] [Unix] [Samba 2.2.8a]
```

Następnym krokiem jest bezpośredni kontakt z serwerami w celu pobrania listy udostępnianych zasobów. Realizację tego zadania umożliwia program `smbclient(8)`, będący interaktywnym klientem protokołu SMB. Wyświetlenie zasobów serwera `unixlab` można więc wykonać następująco:

```
# smbclient -L unixlab
added interface ip=192.168.0.1 bcast=192.168.0.255 nmask=255.255.255.0
Password: ********
Domain=[TUX-NET] OS=[Unix] Server=[Samba 2.2.8a]

        Sharename      Type      Comment
        ---------      ----      -------
        IPC$           IPC       IPC Service (Samba 2.2.8a)
        ADMIN$         Disk      IPC Service (Samba 2.2.8a)
        hp             Printer   HP LaserJet 5MP Postscript
        pub            Disk      Katalog testowy

        Server              Comment
        ---------           -------
        DCS-CSL             Samba 2.2.8a

        Workgroup           Master
        ---------           -------
        TUX-NET
```

Informacje podzielone są na 3 grupy: zasoby udostępniane przez serwer, listę znanych serwerów i listę znanych grup roboczych lub domen. Zasoby, których nazwy kończą się znakiem „$" są zasobami systemowymi. W powyższym przykładzie serwer `unixlab` udostępnia drukarkę pod nazwą „`hp`" oraz katalog pod nazwą „`pub`". Podczas pobierania informacji z serwera pojawia się pytanie o hasło. Puste hasło oznacza dołączenie do serwera w trybie anonimowym. Przełącznik `-N` umożliwia w tym trybie wyłączenie prośby o hasło.

Identyfikacja zdalnych zasobów odbywa się podobnie jak w systemach Windows, np. pełna nazwa katalogu `pub` może być zapisana jako:

```
\\unixlab\pub
```

Znak „\" w interpreterach poleceń systemów Unix ma znaczenie specjalne, stąd musi on być zapisany podwójnie. Możliwe jest również stosowanie znaku „/". Dostęp do zasobu wymaga podłączenia się do niego, co można również zrealizować komendą `smbclient`:

```
# smbclient //unixlab/pub
added interface ip=192.168.0.1 bcast=192.168.0.255 nmask=255.255.255.0
Password: ********
Domain=[TUX-NET] OS=[Unix] Server=[Samba 2.2.8a]
smb: \> help
...
```

Podłączenie do zasobu oznacza przejście do interaktywnego trybu pracy, w którym wykonywane są komendy podobne do tych, z usługi FTP. Oto lista ważniejszych komend programu `smbclient`:

`help` wyświetlenie podręcznej pomocy,

`md/mkdir` utworzenie katalogu,

`rd/rmdir` usuwanie katalogu,

`rm` usuwanie pliku,

`get` pobieranie pliku,

`mget` pobieranie grupy plików,

`put` wstawienie pliku na serwer,

`mput` wstawienie grupy plików,

`prompt` włączenie/wyłączenie potwierdzania przesyłania plików,

`quit` wyjście,

`recurse` przełączenie z/do trybu rekurencyjnego wykonywania komend,

`setmode` odpowiednik DOS-owej komendy `attrib` do manipulacji atrybutami pliku.

Obsługa protokołu SMB jest wbudowana w wiele aplikacji. Przykładem może być domyślna przeglądarka `konqueror` ze środowiska graficznego KDE. Jako adres URL można tam wpisać odwołanie do zasobu z serwera SMB:

```
smb://unixlab/pub
```

Powoduje to zestawienie połączenia do serwera, pobranie nazwy użytkownika i hasła, a następnie wyświetlenie zawartości zdalnego katalogu.

System Linux obsługę protokołu SMB ma wbudowaną w jądro systemu, co pozwala dołączyć zdalny katalog do lokalnej struktury podobnie jak to było wykonywane w przypadku systemu plików NFS:

```
# mount -t smbfs -o username=wojtek //unixlab/pub /mnt
```

Podczas dołączania zdalnego katalogu pojawi się pytanie o hasło, które mogłoby być przekazane dodatkową opcją `password` w linii poleceń. Pełna lista dostępnych opcji znajduje się na stronie pomocy systemowej `smbmount(8)`.

5.3 Konfiguracja serwera

Konfiguracja serwera Samba skupiona jest wokół głównego pliku `smb.conf` zlokalizowanego bezpośrednio w katalogu `/etc` lub w wydzielonym podkatalogu `/etc/samba`. Plik ten składa się z sekcji zawierających parametry. Każda sekcja rozpoczyna się od nazwy umieszczonej w nawiasach kwadratowych i kończy się wraz z początkiem nowej sekcji. Parametry mają składnię *nazwa* = *wartość*. Puste linie i linie zaczynające się od „#" lub „;" są ignorowane (komentarze). Nazwy sekcji i parametrów nie uwzględniają rozróżnienia na duże i małe litery. Linijka kończąca się znakiem „\" jest kontynuowana w następnej. Wartości logiczne reprezentowane są poprzez napisy: 0, 1, `yes`, `no`, `true`, `false`.

Istnieją trzy specjalne sekcje: `[global]`, `[homes]` i `[printers]`. W sekcji `[global]` znajdują się ustawienia dotyczące całego serwera. Jeżeli jakiś parametr może być wykorzystany zarówno w opisie konkretnego zasobu jak i w sekcji globalnej, to umieszczenie jego w sekcji globalnej powoduje ustawienie wartości domyślnej dla wszystkich pozostałych sekcji.

Jeżeli w pliku `smb.conf` jest umieszczona sekcja `[printers]` oznacza to, że użytkownicy mają dostęp do wszystkich lokalnych drukarek skonfigurowanych w pliku `/etc/printcap` (zobacz `printcap(5)`).

Umieszczenie sekcji `[homes]` w pliku `smb.conf` powoduje automatyczne utworzenie oddzielnych sekcji udostępniających katalogi domowe dla każdego lokalnego użytkownika systemu Unix.

5.3.1 Zmienne specjalne

Wartości poszczególnych parametrów konfiguracyjnych mogą się odwoływać do następujących zmiennych specjalnych:

`%D` Nazwa grupy roboczej lub domeny.

`%L` Nazwa komputera do której odwołuje się klient.

`%m` Nazwa komputera w NetBIOS-ie.

`%h` Nazwa komputera (zwracana poleceniem `hostname`).

`%M` Nazwa internetowa komputera.

`%S` Nazwa zasobu, do którego odwołuje się klient.

`%u` Nazwa bieżąca użytkownika.

`%U` Żądana nazwa użytkownika.

`%P` Ścieżka do zasobu.

`%a` Architektura systemu (`Samba`, `WfW`, `WinNT`, `Win95`).

`%I` Adres IP komputera.

`%T` Bieżąca data i czas.

5.3.2 Program administracyjny `swat`

Program `swat` jest serwerem usługi WWW umożliwiającym graficzne zarządzanie serwerem Samba. Uruchomienie usługi wymaga dodania odpowiednich zapisów do konfiguracji serwerów `inetd` lub `xinetd`. W przypadku serwera `inetd` należy dodać następującą linię do pliku `/etc/inetd.conf`:

```
swat stream  tcp nowait.400  root  /usr/sbin/swat  swat
```

W przypadku serwera `xinetd` należy dodać plik konfiguracyjny `/etc/xinetd.d/swat` z następującą zawartością:

```
service swat
{
   socket_type     =  stream
   protocol        =  tcp
   wait            =  no
   user            =  root
   server          =  /usr/sbin/swat
   only_from       =  127.0.0.1
   log_on_failure  += USERID
   disable         =  no
}
```

Po rekonfiguracji należy poinformować serwery o zmianie wykonując jedną z komend:

```
# /etc/init.d/inetd reload
# /etc/init.d/xinetd reload
```

Dostęp do aplikacji `swat` jest możliwy poprzez przeglądarkę pod adresem http://localhost:901/.

5.3.3 Polecenie `smbstatus`

Program `smbstatus` wyświetla informację o wykorzystaniu udostępnianych zasobów. Dostępne przełączniki:

- `-b` wydruk skrócony
- `-d` wydruk rozszerzony
- `-p` lista działających procesów `smbd` związanych z obsługą klienta

Program `testparm` służy do sprawdzenia poprawności zapisu konfiguracji z pliku `/etc/smb.conf`.

5.3.4 Samba a sprawa polska

W sekcji `[global]` można umieścić dodatkowe opcje odpowiedzialne za konwersję kodowania znaków:

```
[global]
   ...
   unix charset = UTF-8
   display charset = ISO8859-2
   dos charset = CP852
```

5.4 Kontroler domeny

5.4.1 Konfiguracja serwera

Poniższy plik `smb.conf` jest przykładową konfiguracją kontrolera domeny (*Primary Domain Controller — PDC*).

```
[global]
   workgroup = TestDomain

   # NetBIOS
   os level = 33
   domain master = yes
   local master = yes
   preferred master = yes

   # obsługa domeny
   domain logons = yes
   # skrypt do logowania użytkownika
   logon script = login.bat
   # katalog z profilem użytkownika
   logon path = \\%L\%U\WinProfile
   # nazwa dysku reprezentującego katalog domowy
   logon drive = H:

[netlogon]
   path = /var/lib/samba/netlogon
   writable = no
   comment = Skrypty logowania

[homes]
   comment = Katalog domowy użytkownika %U
   browseable = no
   writable = yes
```

Przykładowy skrypt `login.bat` logujący użytkowników do systemu:

```
@echo off
net time \\serwer /set /yes
net use H: /home
net use X: \\serwer\programy
start iexplore.exe
```

Skrypt do logowania powinien być poprawnym plikiem tekstowym dla systemów Windows (zakończenia linii). Plik taki można przygotować edytorem `vim`, po wykonaniu komendy `:set fileformat=dos`.

Skonfigurowanie kontrolera domeny wymaga wykonania następujących czynności:

1. Przygotowanie pliku /etc/samba/smb.conf i uruchomienie serwera.
2. Przygotowanie skryptu startowego.
3. Założenie konta *maszynowego* dla każdej stacji roboczej. Konto maszynowe ma taką nazwę jak nazwa komputera z dodanym znakiem „$" na końcu:

   ```
   # useradd 'spica$'
   ```

4. Ustawienie hasła dla administratora sieci SMB na kontrolerze domeny (użytkownik, który ma prawo modyfikować plik z hasłami `/etc/samba/smbpasswd`):

   ```
   # smbpasswd -a root
   ```

5. Dodanie stacji roboczych do domeny poprzez wpisanie nazwy domeny w ustawieniach systemów Windows. Dodanie komputera powoduje zainicjowanie hasła na koncie maszynowym (w pliku `smbpasswd`). Dodanie uniksowej stacji roboczej do domeny wymaga wykonania komendy:

   ```
   # net rpc join -U root
   ```

   gdzie `root` jest administratorem sieci SMB.

6. Założenie kont dla użytkowników. Każdy użytkownik musi mieć zdefiniowane zarówno konto uniksowe, jak i hasło dla protokołu SMB:

   ```
   # useradd user1
   # smbpasswd -a user1
   ```

7. Zdefiniowanie mapowania grup użytkowników na kontrolerze domeny:

   ```
   # net groupmap add unixgroup=users ntgroup="Uzytkownicy PDC"
   ```

   Powyższa komenda spowoduje utworzenie grupy `Uzytkownicy PDC`, której członkami będą użytkownicy należący do uniksowej grupy `users`.

5.4.2 Uwierzytelnianie użytkowników SMB

Użytkownicy protokołu SMB posiadający lokalne konta w systemie Unix, mogą być uwierzytelniani przez kontroler domeny. Wymaga to ustawienia następujących opcji w pliku `/etc/smb.conf`:

```
[global]
   security = server
   password server = 192.168.10.12
   ...
```

Alternatywą jest podłączenie stacji roboczej do domeny (komendą `net rpc`, patrz wyżej) i modyfikacja konfiguracji:

```
[global]
   workgroup = TestDomain
   security = domain
```

Parametr `workgroup` przyjmuje w tym przypadku wartość zgodną z nazwą domeny.

5.4.3 Uwierzytelnianie użytkowników uniksowych

Uwierzytelnianie użytkowników uniksowych może być realizowane za pośrednictwem zewnętrznego serwera PDC udostępniającego hasła użytkowników. Rozwiązanie to wykorzystuje moduł *Pluggable Authentication Module* (PAM) o nazwie `pam_smb`. Konfiguracja wymaga zmian w konfiguracji PAM, w szczególności — w przypadku logowania do systemu — w pliku `/etc/pam.d/login`:

```
#%PAM-1.0
auth     required       pam_securetty.so
auth     required       pam_nologin.so
auth     sufficient     pam_unix2.so
auth     required       pam_smb_auth.so cachetime=20 use_first_pass
...
```

Plik `/etc/pam_smb.conf` zawiera informacje o domenie i nazwach serwerów kontrolujących domenę:

```
UNIXLAB     nazwa domeny
sirius      główny kontroler domeny
spica       zapasowy kontroler domeny
```

5.5 Pakiet Winbind

Pakiet Samba w wersjach 2.2.x i nowszych umożliwia bezpośrednie pobieranie informacji o użytkownikach z serwera PDC bez konieczności tworzenia użytkowników w systemie Unix. Funkcjonalnie jest to rozwiązanie analogiczne do systemu NIS.

Uwaga: podczas konfiguracji oprogramowania `winbind` trzeba wyłączyć usługę `nscd` (buforowanie informacji katalogowych).

5.5.1 Konfiguracja

Konfiguracja stacji roboczej, która ma być dołączona do kontrolera domeny wymaga wykonania następujących czynności:

1. Uzupełnienie konfiguracji w pliku `smb.conf` o ustawienia dla oprogramowania `winbind`:

   ```
   [global]
      ..
      idmap uid = 10000-20000
      idmap gid = 10000-20000
      template homedir = /home/%D/%U
      template shell = /bin/bash
   ```

   gdzie `%D` oznacza nazwę domeny.

2. Wskazanie na źródła informacji o użytkownikach w systemie w pliku `/etc/nsswitch.conf(5)`:

   ```
   passwd:  files winbind
   group:   files winbind
   ```

3. Modyfikacja konfiguracji modułów PAM dla odpowiednich usług. Np., w przypadku logowania interaktywnego należy zmienić zawartość pliku `/etc/pam.d/login`:

   ```
   #%PAM-1.0
   auth     required       pam_securetty.so
   auth     required       pam_nologin.so
   auth     sufficient     pam_unix2.so
   auth     required       pam_winbind.so use_first_pass
   ```

4. Dołączenie stacji roboczej do domeny, z której pobierane są dane (patrz punkt 5.4.1).
5. Uruchomienie procesu usługowego `winbindd`:

```
# winbindd
```

Poprawność pracy serwera `winbind` można sprawdzić programem `wbinfo(1)` pobierającym informacje z tego serwera. Zakładając dołączenie do domeny `unixlab`:

```
# wbinfo -u
UNIXLAB\root
UNIXLAB\user1
# wbinfo -g
...
UNIXLAB\Uzytkownicy PDC
```

Poprawność konfiguracji przełącznika usług katalogowych (plik `/etc/nsswitch.conf`) można sprawdzić wykonując komendę `getent`:

```
# getent passwd
...
UNIXLAB\root
UNIXLAB\user1
```

która powinna zwrócić listę *wszystkich* użytkowników: systemowych i pobranych z PDC. Ostateczną weryfikacją jest możliwość zalogowania się na konto użytkownika sieciowego. Podczas logowania należy podać całą nazwę użytkownika, łącznie z domeną, np.: `UNIXLAB\user1`.

Odwoływanie się do użytkowników domenowych może być uproszczone poprzez przyjęcie domyślnej nazwy domeny przez oprogramowanie `winbind`. Wymaga to ustawienia dodatkowej opcji w sekcji `[global]`:

```
winbind use default domain = yes
```

Pozwala to na podawanie nazwy użytkowników z pominięciem nazwy domeny, czyli np. `user1`.

5.5.2 Katalogi domowe użytkowników uniksowych

Katalogi domowe użytkowników mogą być automatycznie dołączane z kontrolera domeny za pośrednictwem modułu PAM o nazwie `pam_mount`[1]. Wymaga go konfiguracji odpowiedniego modułu PAM, np. w przypadku logowania interaktywnego będzie to plik `/etc/pam.d/login`:

```
#%PAM-1.0
auth      required    pam_securetty.so
auth      required    pam_nologin.so
auth      optional    pam_mount.so
auth      sufficient  pam_winbind.so  use_first_pass
auth      required    pam_unix2.so    use_first_pass

session   required    pam_unix2.so    none
session   optional    pam_mount.so
```

[1]Moduł można pobrać ze strony http://pam-mount.sourceforge.net/.

Moduł pam_mount wymaga również wskazania katalogów, które powinny zostać dołączone. Wskazanie to jest zapisane w pliku /etc/security/pam_mount.conf:

```
volume * smb unixlab & /home/& \
uid=&,gid=students,dmask=0700,fmask=0700,workgroup=TestDomain - -
```

Alternatywnym rozwiązaniem jest tworzenie katalogów domowych „w locie" z wykorzystaniem modułu pam_mkhomedir. Konfiguracja modułu wymaga dodania do odpowiedniego pliku z katalogu /etc/pam.d zapisu:

```
session   required     pam_mkhomedir.so skel=/etc/skel/ umask=0022
```

5.6 Opis parametrów konfiguracyjnych

Przy nazwie każdego parametru umieszczono literę G — oznaczającą, że dany parametr może być użyty jedynie w sekcji [global] lub S — oznaczającą, że parametr może być wykorzystany zarówno do opisu zasobu jak i w sekcji [global].

Parametry ogólne

netbios name (G)
: Nazwa serwera. Domyślnie przyjmowana jest nazwa zwracana poleceniem hostname.

server string (G)
: Opis serwera: %v oznacza wersję Samby, %h — nazwę *hostname*.

character set (G)
: Standard kodowania wykorzystywany przez serwer, np. iso8859-2.

client code page (G)
: Strona kodowa systemu plików DOS–a. Numer strony kodowej można sprawdzić w DOS–ie poleceniem chcp. Dla języka polskiego należy ustawić stronę kodową 852.

max connections (S)
: Maksymalna liczba współbieżnych połączeń klientów. Zero oznacza brak ograniczeń.

time server (G)
: Wskazuje, że proces nmbd będzie serwerem czasu dla systemów Windows.

Opis zasobu

path (S)
: Ścieżka określająca zasób. Dla drukarek jest to nazwa katalogu, do którego będą zapisywane zlecenia wydruku (ang. *spool*).

admin users (S)
: Lista administratorów usług. Wszyscy użytkownicy wymienieni na tej liście będą mogli wykonywać wszelkie operacje dotyczące danej usługi, bez względu na uprawnienia.

browsable (S)
: Włącza widoczność zasobu na liście dostępnych zasobów.

`writable` (S)
: Zezwala na zapis do katalogu.

`printable` (S)
: Określa zasób jako drukarkę.

`comment` (S)
: Opis widoczny na liście zasobów.

Prawa dostępu

`security` (G)
: Sposób weryfikacji praw dostępu do zasobów. Możliwe wartości: `share`, `user`, `server`, `domain`. Tryb `user` (domyślny) może być stosowany jeżeli w systemie Unix mają swoje konta użytkownicy innych systemów ubiegających się o dostęp. Tryb `share` należy wykorzystać w przypadku braku takich kont lub przy dostępie publicznym (bez weryfikacji hasła). Tryby `server` i `domain` oznaczają, że Samba kontaktuje się z kontrolerem domeny w celu uwierzytelnienia użytkownika.

`public` (S)
: Zasób publiczny nie wymagający hasła podczas dostępu.

`hosts allow` (S)
: Lista komputerów/sieci, które mogą korzystać z zasobu.

`hosts deny` (S)
: Lista komputerów/sieci, dla których zasób jest niedostępny. Jeżeli występują konflikty pomiędzy `hosts allow` a `hosts deny`, to zawsze priorytet ma `hosts allow`.

`create mask` (S)
: Maska praw, co najwyżej które będą ustawione przy tworzeniu nowych plików:

```
create mask = 700
```

`force create mode` (S)
: Prawa, które zostaną dodane do praw dostępu do nowo tworzonych plików.

`directory mask` (S)
: Analogicznie do `create mask`, ale w stosunku do katalogów.

`force directory mode` (S)
: Analogicznie jak `force create mode`, ale w odniesieniu do katalogów.

`follow symlinks` (S)
: Zezwala na przechodzenie do katalogów wskazywanych przez dowiązania symboliczne.

`force user` (S)
: Wymuszenie zmiany nazwy użytkownika przy dostępie do zasobu.

`force group` (S)
: Wymuszenie zmiany nazwy grupy przy dostępie do zasobu.

`guest account` (S)
: Konto użytkownika anonimowego.

`hide dot files` (S)
: Prezentacja plików zaczynających się od kropki jako plików ukrytych.

`hide files` (S)
: Lista plików, które powinny zostać wyświetlone jako ukryte:

```
hide files = /.*/Trash/*~/
```

`only user` (S)
: Zabrania logowania użytkowników spoza listy.

`read list` (S)
: Lista użytkowników, którzy mają tylko prawo do odczytu, bez względu na ustawienie parametru `writable`. Można dołączać całe grupy za pomocą notacji @*grupa*. Przykład:

```
read list = john, @students
```

`invalid users` (S)
: Lista użytkowników niedopuszczonych do zasobu. Nazwa rozpoczynająca się od `@` reprezentuje grupę sieciową lub grupę uniksową, jeżeli nie znaleziono odpowiedniej grupy sieciowej. Nazwa rozpoczynająca się od `+` reprezentuje tylko i wyłącznie grupę uniksową. Nazwa rozpoczynająca się od `&` reprezentuje tylko i wyłącznie grupę sieciową.

`valid users` (S)
: Lista użytkowników, którzy mają dostęp do zasobu. Przykład:

```
valid users = fred, @staff
```

Jeżeli użytkownik występuje na liście `valid users` i `invalid users`, to dostęp do zasobu jest dla niego zabroniony.

Użytkownicy

`logon home` (G)
: Katalog domowy dla klientów logujących się z systemów Windows. Umożliwia to wykonanie na tych komputerach komendy:

```
C:\> NET USE H: /HOME
```

definiującej dysk `H:` jako wskazujący na katalog domowy. Domyślne ustawienie:

```
logon home = "\\%N\%U"
```

`logon path` (G)
: Katalog z plikami konfiguracyjnymi użytkownika (profil) dla systemów NT/2000/XP. Domyślne ustawienie:

```
logon path = \\%N\%U\profile
```

`logon script` (G)
: Skrypt, który będzie wykonywany po zalogowaniu się do domeny kontrolowanej przez Sambę.

`username map` (G)
: Plik mapujący nazwy użytkowników protokołu SMB na użytkowników Uniksa. Przykład takiego pliku:

```
root = admin administrator "Marek Kowalski"
sys = @system
guest = *
```

`domain logons` (G)
: Określa serwer jako kontroler domeny (PDC).

`encrypt passwords` (G)
: Windows od wersji NT 4.0 SP3 oraz Windows 98 i nowsze stosują kodowanie haseł.

`password server` (G)
: Nazwa serwera, który będzie autoryzował użytkowników. Wymaga również ustawienia parametru `security` na wartość `server` lub `domain`.

Komunikaty WinPopup

`message command` (G)
: Nazwa komendy, która wyświetli komunikat WinPopup. Przykład:

```
message command = sh -c 'xedit %s; rm -f %s' &
```

Komenda powinna natychmiast zwracać sterowanie do systemu. `%s` reprezentuje plik z wiadomością, `%t` — adresata, `%f` — nadawcę.

Wysyłanie komunikatów:

`smbclient -M` *hostname*

Należy wprowadzać tekst i zakończyć go poprzez Ctrl–D. Można również informację przekazać przez standardowe wejście:

`ps -ax | smbclient -M` *hostname*

Drukarki

Pole `path` w przypadku drukarek określa katalog, w którym będą tworzone kopie lokalne przesłanych do wydruku plików. Katalog ten powinien być dostępny do zapisu dla wszystkich i mieć ustawiony bit *sticky* (`chmod +t`).

`load printers` (G)
: Parametr określający czy mają być udostępniane wszystkie skonfigurowane lokalnie drukarki (plik `/etc/printcap`).

`printing` (S)
: Specyfikuje rodzaj podsystemu drukowania. Komenda ta ustawia również wartości domyślne dla ustawień: `print command`, `lpq command`, `lprm command`.

`print command` (S)
Komenda, która zostanie użyta do wydrukowania przesłanego przez klienta pliku. Można wykorzystać następujące zmienne specjalne: `%s` reprezentuje nazwę lokalnej kopii zleconego do wydruku pliku, `%f` — to ta sama nazwa, ale bez pełnej ścieżki, `%p` — to nazwa drukarki.

`lpq command` (S)
Komenda wykonywana w celu uzyskania statusu drukarki.

`lprm command` (S)
Komenda usuwająca zadanie drukowania z kolejki. Można wykorzystać specjalne zmienne: `%p` — nazwa drukarki, `%j` — numer zlecenia. Przykład:

```
lprm command = /usr/bin/lprm -P %p %j
```

Diagnostyka

`log level` (G)
Stopień szczegółowości informacji diagnostycznych zapisywanych w pliku log.

`log file` (G)
Nazwa pliku, do którego zapisywane są informacje diagnostyczne.

`syslog` (G)
Określa stopień szczegółowości komunikatów przekazywanych do systemowej usługi `syslog`

`syslog only` (G)
Wskazuje, że komunikaty diagnostyczne powinny być przekazywane tylko do usługi `syslog`, czyli nie do plików `log file`.

Dodatkowe informacje

1. Pomoc systemowa `man`: `samba(7)`, `smb.conf(5)`, `smbd(8)`, `nmbd(8)`, `smbmount(8)`, `printcap(5)`, `winbind(8)`.
2. Strona WWW: http://www.samba.org/
3. NetBIOS: RFC 1001, RFC 1002.
4. *Using Samba* [ECBK03]. Książka dołączona do dokumentacji pakietu Samba.

Zadania

1. Zlokalizuj zdalne zasoby udostępniane w sieci lokalnej poprzez uruchomienie serwera `nmbd` i korzystając z programów `findsmb` oraz `smbclient`.
2. Wykonaj dostęp do zdalnego zasobu za pośrednictwem programu `smbclient`, przeglądarki Konqueror oraz poprzez montowanie zdalnego katalogu.
3. Skonfiguruj i uruchom serwer udostępniający przykładowy katalog do zapisu. Zweryfikuj możliwość dokonywania modyfikacji w tym katalogu.

4. Sprawdź poprawność odwzorowania polskich znaków w nazwach plików. Pliki tworzone po stronie systemu Windows powinny być poprawnie widziane po stronie systemu Unix i odwrotnie.
5. Przetestuj uwierzytelnianie użytkowników w oparciu o zewnętrzny serwer.
6. Ogranicz dostępność katalogu dla wybranych komputerów, użytkowników lub grup sieciowych.
7. Zmień konfigurację praw dostępu dla nowotworzonych plików, tak aby członkowie grupy mieli możliwość zapisu, a pozostali użytkownicy nie posiadali żadnych praw.
8. Przetestuj działanie programu zarządzającego `swat`.
9. Skonfiguruj kontroler domeny i dołącz do niego system Windows oraz Unix. Przetestuj działanie wędrujących profili użytkowników (ang. *roaming profiles*) logując się do różnych stacji Windows w tej samej domenie. Przetestuj działanie skryptu logowania.
10. Uruchom usługę Winbind na stacji Unix importując konta użytkowników z kontrolera domeny. Skonfiguruj dostęp do katalogów domowych użytkowników.

6 Network Information Service

6.1 Wprowadzenie

NIS (ang. *Network Information Service*) jest usługą katalogową, której głównym zadaniem jest umożliwienie współdzielenia informacji konfiguracyjnych w sieci. Jednym z podstawowych zastosowań jest rozprowadzanie informacji o użytkownikach systemu. Poprawnie skonfigurowany system umożliwia zrealizowanie scentralizowanego zarządzania bazą danych użytkowników.

System NIS działa w architekturze klient-serwer. Klient pobiera informacje katalogowe z serwera poprzez biblioteki systemowe, a więc w sposób przezroczysty dla aplikacji.

W jednej sieci lokalnej może pracować kilka serwerów NIS przeznaczonych dla różnych klientów. Przypisanie klientów do serwerów odbywa się za pośrednictwem tzw. *domen*. Pojedynczy serwer może obsługiwać wiele domen, podobnie klient może należeć do wielu. Domeny systemu NIS są mechanizmem całkowicie niezależnym od domen systemu DNS (mogą się one dowolnie przenikać). W systemie NIS domeny są płaskie, a więc nie jest możliwe tworzenie struktur hierarchicznych.

Informacje w systemie NIS są reprezentowane w postaci tzw. *map*, czyli baz danych przygotowywanych z istniejących w systemie plików konfiguracyjnych. Każda mapa zawiera rekordy identyfikowane pewnym kluczem. Przykładem może być baza danych o użytkownikach (zawartość pliku `/etc/passwd`) posortowana wg ich identyfikatorów:

```
+-------+-----------------------------------------------------------------+
| Klucz | Wartość                                                         |
+-------+-----------------------------------------------------------------+
|  1000 | sobaniec:x:1000:100:Cezary Sobaniec:/home/sobaniec:/bin/bash    |
|  1001 | kaminski:x:1001:100:Wojciech Kaminski:/home/kaminski:/bin/bash |
+-------+-----------------------------------------------------------------+
```

6.2 Konfiguracja serwera

Konfigurację serwera należy rozpocząć od ustalenia domyślnej domeny dla systemu NIS:

```
# domainname nistest
```

Polecenie to działa analogicznie do polecenia `hostname(1)`. To samo zadanie realizują polecenia `nisdomainname(8)` i `ypdomainname(8)`. Polecenie `domainname(8)` uruchomione bez argumentów wyświetla nazwę domyślnej domeny:

```
# domainname
nistest
```

Następnym krokiem jest uruchomienie serwera:

```
# ypserv
```

Usługa NIS implementowana jest z wykorzystaniem mechanizmu RPC. Uruchomienie jej powoduje więc zarejestrowanie nowych usług:

```
# rpcinfo -p
   program vers proto   port
    100000    2   tcp    111  portmapper
    100000    2   udp    111  portmapper
    ...
    100004    2   udp    830  ypserv      ------+
    100004    1   udp    830  ypserv            | nowe
    100004    2   tcp    833  ypserv            | usługi
    100004    1   tcp    833  ypserv      ------+
    ...
```

Pliki konfiguracyjne systemu NIS mieszczą się w katalogu /var/yp[1]. Początkowa zawartość tego katalogu jest zbliżona do poniższego przykładu:

```
/var/yp/
  |
  |-- Makefile
  |-- binding/
  |     |
  |     |-- nistest.1
  |     `-- nistest.2
  |
  |-- nicknames
  `-- securenets
```

Plik Makefile zawiera specyfikacje dla programu make(1) zarządzającego kompilacją map nisowych. W katalogu /binding znajdują się pliki tworzone przez klienta usługi NIS w momencie nawiązania łączności z odpowiednim serwerem. Pliki te mają nazwy takie, jak nazwa domeny i rozszerzenie wskazujące na wersję systemu NIS. Plik nicknames zawiera odwzorowania nazw skróconych dla map nisowych (aliasy), a plik securenets umożliwia ograniczenie dostępu do serwera NIS.

Konfiguracja serwera NIS sprowadza się do wskazania jakie mapy nisowe mają być utworzone i na podstawie jakich plików. Ustawienia te podaje się w pliku Makefile. Linia zaczynająca się od słowa all definiuje jakie mapy nisowe będą tworzone. Standardowo istnieje możliwość tworzenia m.in. map nisowych wymienionych w tabeli 6.1.

Po dokonaniu modyfikacji pliku Makefile można utworzyć mapy nisowe serwera:

```
# make
gmake[1]: Entering directory `/var/yp/nistest'
Updating passwd.byname...
Updating passwd.byuid...
Updating group.byname...
```

[1] Początkowo system NIS zwany był *Yellow Pages*. Nazwa ta została później zastrzeżona, ale w nazwach programów i plików pozostał prefiks yp.

Tab. 6.1: Mapy systemu NIS

Nazwa mapy	Plik konfiguracyjny	Opis
passwd	/etc/passwd(5)	Baza danych użytkowników
shadow	/etc/shadow(5)	Hasła użytkowników
group	/etc/group(5)	Definicje grup użytkowników
hosts	/etc/hosts(5)	Lokalne odwzorowania nazw na adresy IP
services	/etc/services(5)	Odwzorowania nazw usług na numery portów
rpc	/etc/rpc(5)	Odwzorowania nazw usług RPC na numery
netgroup	/etc/netgroup(5)	Definicje grup sieciowych
auto.master	/etc/auto.master(5)	Konfiguracja automontera (zobacz punkt 4.5)
auto.home	/etc/auto.home(5)	Konfiguracja katalogu /home dla automontera

```
Updating group.bygid...
Updating services.byname...
Updating services.byservicename...
Updating hosts.byname...
Updating hosts.byaddr...
gmake[1]: Leaving directory '/var/yp/nistest'
```

Wykonanie programu `make` powoduje utworzenie katalogu takiego, jak nazwa domeny i utworzenie w nim map nisowych. Dla niektórych plików konfiguracyjnych tworzone są dwie mapy nisowe, indeksowane różnymi kluczami. Przykładowo: baza danych użytkowników indeksowana jest po identyfikatorach numerycznych (mapa `passwd.byuid`) i nazwach (mapa `passwd.byname`).

Poprawność konfiguracji możemy sprawdzić programem `ypcat(1)`, który umożliwia wyświetlanie całych baz danych serwera:

```
# ypcat -h localhost passwd
kaminski:x:1001:100:Wojciech Kaminski:/home/kaminski:/bin/bash
sobaniec:x:1000:100:Cezary Sobaniec:/home/sobaniec:/bin/bash
```

Dane można wyświetlać łącznie z kluczem indeksującym, co pozwala na zaobserwowanie różnic pomiędzy wersjami map powstałymi z tego samego pliku:

```
# ypcat -h localhost -k passwd.byname
kaminski kaminski:x:1001:100:Wojciech Kaminski:/home/kaminski:/bin/bash
sobaniec sobaniec:x:1000:100:Cezary Sobaniec:/home/sobaniec:/bin/bash
...
# ypcat -h localhost -k passwd.byuid
1000 sobaniec:x:1000:100:Cezary Sobaniec:/home/sobaniec:/bin/bash
1001 kaminski:x:1001:100:Wojciech Kaminski:/home/kaminski:/bin/bash
...
```

6.3 Konfiguracja klienta

Podobnie jak w serwerze na początku należy ustawić nazwę domeny. Konfiguracja klienta sprowadza się do wskazania serwera w pliku `/etc/yp.conf`:

```
ypserver unixlab.cs.put.poznan.pl
```

Adres serwera w lokalnej sieci może być również automatycznie odszukany poprzez rozgłoszenie:

```
broadcast
```

Wskazanie wielu serwerów dla wielu domen jest możliwe dzięki słowu kluczowemu `domain`:

```
domain nistest server galio.cs.put.poznan.pl
domain nistest2 broadcast
```

W przykładzie dane z domeny `nistest` pochodzą z serwera `galio`, a dane z domeny `nistest2` z komputera w lokalnej sieci.

Klient NIS wymaga uruchomienia programu usługowego `ypbind(8)`, którego zadaniem jest utrzymywanie łączności z serwerem i pobieranie od niego informacji. Należy więc wykonać komendę:

```
# ypbind
```

Odnalezienie klienta dla wskazanej domeny i poprawne dołączenie się do niej powinno spowodować powstanie w katalogu `/var/yp/binding` pliku o nazwie złożonej z nazwy domeny i numeru wersji systemu NIS. Poprawne dołączenie do serwera można sprawdzić komendą `ypwhich(8)`:

```
# ypwhich
galio.cs.put.poznan.pl
# ypwhich -d nistest2
unixlab.cs.put.poznan.pl
```

Klient po nawiązaniu łączności z serwerem może pobierać z niego dane:

```
# ypcat passwd
...
```

Pobieranie pojedynczych rekordów z mapy nisowej umożliwia komenda `ypmatch(8)`. Wymaga ona podania klucza i nazwy mapy nisowej:

```
# ypmatch sobaniec passwd.byname
sobaniec:x:1000:100:Cezary Sobaniec:/home/sobaniec:/bin/bash
# ypmatch 1000 passwd.byuid
sobaniec:x:1000:100:Cezary Sobaniec:/home/sobaniec:/bin/bash
```

Pomimo nawiązania łączności z serwerem i dostępności informacji np. o użytkownikach, dane te nie są jeszcze wykorzystywane przez system. Wymaga to jawnego wskazania serwera NIS jako źródła informacji katalogowej. Dokonuje się tego w pliku konfiguracyjnym `/etc/nsswitch.conf(5)`:

```
passwd:   files nis
group:    files nis
```

Powyższe linie spowodują, że dane o użytkownikach i grupach będą najpierw pobierane z lokalnych plików konfiguracyjnych (`/etc/passwd` i `/etc/group`), a później z serwera NIS (z map `passwd` i `group`).

6.4 Zmiana hasła użytkownika

System NIS służy do propagacji informacji konfiguracyjnych w trybie tylko do odczytu. W większości przypadków jest to rozwiązanie wystarczające, gdyż rekonfiguracja wymaga uprawnień administratora. Hasła użytkowników są tutaj wyjątkiem, który wymaga specjalnego traktowania. Standardowe polecenie `passwd(1)` dokonuje zmiany lokalnego hasła. W systemie, który korzysta z konfiguracji dostarczonej za pośrednictwem usługi NIS, zmiana hasła przekazywana jest do serwera za pośrednictwem dodatkowej usługi RPC. Po stronie serwera należy uruchomić program:

```
# rpc.yppasswdd
```

Zmiana hasła po stronie klienta może zostać wykonana standardowym poleceniem `passwd(1)` lub `yppasswd(1)`. Istnieją również odpowiedniki komend `chsh(1)` i `chfn(1)` dla systemu NIS o nazwach odpowiednio `ypchsh(1)` i `ypchfn(1)`.

Zmiana hasła po stronie serwera powoduje aktualizację bazy danych o użytkowniku (pliki `/etc/passwd` i `/etc/shadow`) i jednocześnie aktualizację odpowiednich map. Zmiana hasła na serwerze, ale wykonana standardowym poleceniem `passwd(1)`, powoduje zmianę jedynie lokalnych plików. Aktualizację map nisowych można wykonać wykonując samodzielnie komendę `make`:

```
# make -C /var/yp
```

6.5 Serwery pomocnicze

W przypadku dużych sieci korzystne jest uruchomienie kilku serwerów NIS, które będą współpracowały w obsługiwaniu klientów. Współpraca realizowana jest poprzez wprowadzenie replikacji informacji z serwera głównego. Serwery podrzędne powielają informacje uzyskane od serwera głównego. Serwer podrzędny udziela dokładnie tych samych informacji co serwer główny i jest traktowany identycznie przez klientów jak serwer główny.

Uruchomienie serwera pomocniczego wymaga wstępnego skonfigurowania i uruchomienia serwera głównego. Można do tego celu wykorzystać skrypt `ypinit(8)`:

```
# /usr/lib/yp/ypinit -m
```

Skrypt ten utworzy plik `/var/yp/ypservers` zawierający listę serwerów obsługujących daną domenę. Następnie na planowanym serwerze pomocniczym należy ustawić domenę NIS i wykonać komendę `ypinit(8)`:

```
# /usr/lib/yp/ypinit -s unixlab.cs.put.poznan.pl
```

gdzie nazwa po przełączniku `-s` jest nazwą serwera głównego NIS. Wykonanie komendy powoduje utworzenie katalogu o nazwie domeny w katalogu `/var/yp` i skopiowanie do niego wszystkich map z serwera głównego. Po uruchomieniu programu `ypserv`, serwer pomocniczy może już udostępniać informacje klientom, którzy będą się ubiegali o te informacje.

Propagacja informacji do serwerów pomocniczych odbywa się przy pomocy programu `yppush(8)`. Po każdej zmianie wykonanej na serwerze, podczas wykonywania komendy `make`, informacja o nowych mapach nisowych jest „wypychana" do serwerów pomocniczych. Lista serwerów pomocniczych znajduje się w pliku `/var/yp/ypservers`. Serwery

pomocnicze po odebraniu powiadomienia o zmianie na serwerze głównym sprowadzają nową wersję mapy. Informowanie serwerów pomocniczych wymaga ustawienia zmiennej `NOPUSH` na wartość `false` w pliku `/var/yp/Makefile` na serwerze głównym. Aktualizowanie informacji na serwerach pomocniczych może być przyspieszone dzięki wykorzystaniu programu `rpc.ypxfrd(8)`, który przesyła całą mapę jako plik nie wymagając jej rekonstrukcji po stronie serwera pomocniczego. Program powinien być uruchomiony po stronie serwera.

Uwaga: aktualizacja map na serwerze głównym wymaga, aby posiadał on uruchomiony proces klienta `ypbind`.

6.6 Współpraca NIS i NFS

System NIS może być wykorzystywany łącznie z systemem NFS. Identyfikacja użytkowników w systemie NFS odbywa się poprzez ich identyfikatory numeryczne, co stwarza problemy w przypadku niespójności baz danych użytkowników. Taką spójność może zapewnić właśnie system NIS. Serwery NFS i NIS mogą być uruchomione w ramach jednego systemu lub różnych.

6.7 Bezpieczeństwo

System NIS, podobnie jak system plików NFS, nie należy do zbyt bezpiecznych, chociażby z powodu wydostawania się zakodowanych haseł poza bazowy system operacyjny. Z tego powodu należy go stosować tylko w zamkniętych, wyizolowanych sieciach lokalnych. Jednocześnie należy ograniczyć liczbę komputerów, które będą mogły odwoływać się do serwera. Ograniczenia takie można wyrazić w pliku `/var/yp/securenets(5)`. Oto przykład zawartości tego pliku konfiguracyjnego:

```
# Dostęp z adresu 127.0.0.1
255.0.0.0 127.0.0.0

# Dostęp dla wybranego komputera
host 192.168.10.5

# Dostęp dla wszystkich
0.0.0.0   0.0.0.0
```

Więcej informacji nt. systemu NIS można uzyskać na stronach pomocy systemowej: `ypserv(8)`, `ypbind(8)`, `ypcat(1)`, `yppush(8)`, `rpc.yppasswdd(8)`, `ypxfr(8)`, `rpc.ypxfrd(8)`, `passwd(5)`, `shadow(5)`.

Zadania

1. Znajdź serwer NIS dla domyślnej domeny stacji roboczej.
2. Pobierz informacje o użytkownikach z serwera za pomocą komend `ypcat` i `ypmatch`.
3. Skonfiguruj serwer NIS dla domeny o wybranej nazwie, udostępniający informacje o użytkownikach, grupach, grupach sieciowych i nazwach komputerów.
4. Skonfiguruj oprogramowanie automontera pobierając konfigurację z serwera NIS.

5. Skonfiguruj serwery NIS i NFS udostępniające informacje o użytkownikach i ich katalogi domowe dla stacji roboczej. Serwer NIS powinien pozwalać na zmianę hasła użytkowników z poziomu stacji roboczej.
6. Skonfiguruj serwer pomocniczy dla serwera NIS.
7. Dołącz stację roboczą do dwóch różnych domen NIS.
8. Ogranicz prawo dostępu do serwera NIS tylko dla lokalnej podsieci.
9. Na serwerze NIS, w pliku `/etc/hosts` dopisz definicję nowego komputera, np. `nishost`. Następnie sprawdź możliwość odwzorowania tej nazwy na adres IP po stronie klienta systemu NIS, wykonując np.:

   ```
   # getent hosts nishost
   ```

   Dlaczego nie jest możliwie odwzorowanie tej nazwy z użyciem komendy `host`?

7 LDAP

7.1 Konfiguracja serwera

Ćwiczenia niniejsze zakładają użycie serwera OpenLDAP. Na początek należy serwer skonfigurować i uruchomić. Dane serwera przechowywane są standardowo w katalogu `/var/lib/ldap`. Tworząc nową instalację należy usunąć całą zawartość tego katalogu.

Następnym krokiem jest wygenerowanie hasła dla administratora. Realizuje to komenda `slappasswd(8)`:

```
# slappasswd
New password: ********
Re-enter new password: ********
{SSHA}Bcq2BUtjJPPORHIXLYSZgYgsHRAROErm
```

Hasło to należy wpisać do pliku konfiguracyjnego serwera OpenLDAP `/etc/openldap/slapd.conf`. W pliku tym powinny się również znaleźć inne ustawienia wymienione poniżej:

```
include         /etc/openldap/schema/core.schema
include         /etc/openldap/schema/cosine.schema
include         /etc/openldap/schema/inetorgperson.schema
include         /etc/openldap/schema/rfc2307bis.schema
...
loglevel        256
...
database        bdb
suffix          "dc=put,dc=pl"
rootdn          "cn=admin,dc=put,dc=pl"
rootpw          {SSHA}Bcq2BUtjJPPORHIXLYSZgYgsHRAROErm
```

Komenda `include` załącza zewnętrzne pliki konfiguracyjne. W tym przypadku załączane są pliki zawierające definicje schematów danych (zobacz niżej). Komenda `loglevel` ustawia poziom szczegółowości rejestracji zdarzeń zachodzących w serwerze. Wartość 256 jest polem bitowym aktywującym rejestrację wykonywanych operacji protokołu LDAP. Od komendy `database` rozpoczyna się definicja repozytorium danych serwera. Opcja `suffix` definiuje sufiks dla wszystkich obiektów przechowywanych w tym repozytorium. Administratorem tych danych będzie użytkownik identyfikowany jako `cn=admin,dc=put,dc=pl`.

Pozostałe ustawienia konfiguracyjne opisane są na stronie pomocy systemowej `slapd.conf(5)`.

Schematy

Serwer OpenLDAP rozprowadzany jest łącznie z definicjami wielu standardowych schematów danych. Poniżej wymieniono te najważniejsze:

`core.schema`
: Podstawowe atrybuty LDAPv3 i klasy opisane w RFC 2251–2256.

`cosine.schema`
: Podstawowe atrybuty i klasy z projektu COSINE opisane w RFC 4524.

`inetorgperson.schema`
: Opis klasy `inetOrgPerson` i jej atrybutów zdefiniowanych w RFC 2798.

`nis.schema`, `rfc2307bis.schema`
: Klasy i atrybuty potrzebne do integracji z systemem NIS. Opis w RFC 2307.

`samba3.schema`
: Klasy i atrybuty do pracy serwera Samba.

`java.schema`
: Schemat opisany w RFC 2713 dla zapisu strumieniowego obiektów Java.

`openldap.schema`
: Definicje klas serwera OpenLDAP.

`misc.schema`
: Różne definicje uzupełniające (również eksperymentalne).

`corba.schema`
: Schemat do przechowywania obiektów Corby. Opis w RFC 2714.

7.2 Edycja rekordów

7.2.1 Podstawowe narzędzia

Po uruchomieniu serwera baza danych jest pusta, nie zawiera nawet obiektów, do których odnoszą się zapisy w głównym pliku konfiguracyjnym. Pierwszym ćwiczeniem musi więc być wykonanie dodawania rekordów do bazy informacji katalogowej. Operacje na bazie wykonywane są zawsze za pośrednictwem formatu LDIF. Poniższy plik `base.ldif` zawiera podstawowe obiekty przykładowego serwera LDAP:

```
dn:             dc=put,dc=pl
objectclass:    dcObject
objectclass:    organization
o:              Poznan University of Technology
dc:             put

dn:             cn=admin,dc=put,dc=pl
objectclass:    organizationalRole
cn:             admin

dn:             ou=people,dc=put,dc=pl
ou:             people
```

```
objectClass:      organizationalUnit
description:      Uzytkownicy

dn:               ou=groups,dc=put,dc=pl
ou:               groups
objectClass:      organizationalUnit
description:      Grupy uzytkownikow

dn:               ou=hosts,dc=put,dc=pl
ou:               hosts
objectClass:      organizationalUnit
description:      Komputery
```

Dodawanie rekordów do serwera LDAP umożliwia komenda `ldapadd(1)`:

```
# ldapadd -x -W -h localhost -D cn=admin,dc=put,dc=pl -f base.ldif
Enter LDAP Password: ****
adding new entry "dc=put,dc=pl"

adding new entry "cn=admin,dc=put,dc=pl"

adding new entry "ou=people,dc=put,dc=pl"

adding new entry "ou=groups,dc=put,dc=pl"

adding new entry "ou=hosts,dc=put,dc=pl"
```

Wywołanie programu `ldapadd` zawiera wskazanie na komputer `localhost`, gdzie uruchomiony jest serwer LDAP. Domyślnie przyjmowany jest serwer wskazany w pliku `/etc/openldap/ldap.conf` (dalsze przykłady zakładają ustawienie tych parametrów):

```
host            sirius.cs.put.poznan.pl
base            dc=put,dc=pl
```

Pobranie rekordów z serwera wykonuje komenda `ldapsearch(1)`:

```
# ldapsearch -x -b dc=put,dc=pl '(objectClass=*)'
# extended LDIF
#
# LDAPv3
# base <dc=put,dc=pl> with scope sub
# filter: (objectClass=*)
# requesting: ALL
#

# put.pl
dn: dc=put,dc=pl
objectClass: dcObject
objectClass: organization
o: Poznan University of Technology
dc: put

# admin, put.pl
dn: cn=admin,dc=put,dc=pl
objectClass: organizationalRole
cn: admin
```

```
...
```

Definicja przykładowego użytkownika:

```
dn:              uid=wojtek,ou=people,dc=put,dc=pl
objectClass:     account
objectClass:     posixAccount
uid:             wojtek
cn:              Kowalski
uidNumber:       2123
gidNumber:       110
homeDirectory:   /home/wojtek
userPassword:    qwerty7
description:     Uzytkownik Wojtek
loginShell:      /bin/bash
```

Definicja przykładowej grupy:

```
dn:              cn=staff,ou=groups,dc=put,dc=pl
objectClass:     namedObject
objectClass:     posixGroup
cn:              staff
gidNumber:       110
```

Oto przykłady innych zleceń wyszukujących atrybuty obiektów z bazy informacji katalogowej:

```
# ldapsearch -x -b dc=put,dc=pl '(objectClass=organizationalUnit)' ou
# ldapsearch -x -b ou=people,dc=put,dc=pl -s one '(objectClass=*)'
# ldapsearch -x -b dc=put,dc=pl '(&(objectClass=posixAccount)(uid=wojtek))'
```

Dodawanie i usuwanie atrybutów w bazie danych wymaga również przygotowania odpowiedniego pliku w formacie LDIF. Poniższy przykład powoduje usunięcie atrybutu `description` z obiektu `uid=wojtek,ou=people,dc=put,dc=pl`:

```
dn:              uid=wojtek,ou=people,dc=put,dc=pl
changetype:      modify
delete:          description
```

Zawartość odpowiedniego pliku LDIF należy przetworzyć programem `ldapmodify(1)`:

```
# ldapmodify -x -W -D cn=admin,dc=put,dc=pl -f delete.ldif
modifying entry "uid=wojtek,ou=people,dc=put,dc=pl"
```

Modyfikacja atrybutów wymaga ich usunięcia i późniejszego dodania:

```
dn:              uid=wojtek,ou=people,dc=put,dc=pl
changetype:      modify
delete:          cn
-
add:             cn
cn:              Wojciech Kowalski
```

Usuwanie całych obiektów realizowane jest komendą `ldapdelete(1)`:

```
# ldappdelete -x -W -D cn=admin,dc=put,dc=pl \
              uid=wojtek,ou=people,dc=put,dc=pl
```

7.2.2 Dostęp poprzez przeglądarkę

Wyszukiwanie danych z serwera LDAP można również realizować poprzez przeglądarkę, posiłkując się specjalnie przygotowanymi adresami URL wg. poniższego schematu:

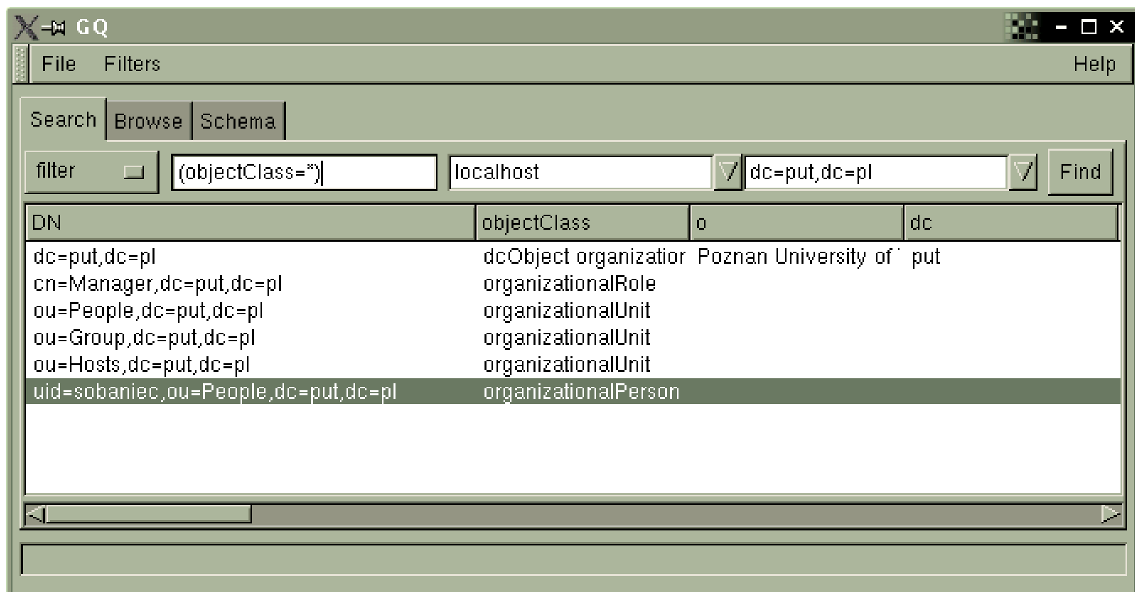
`ldap://`*hostname*`:`*port*`/`*base_dn*`?`*attributes*`?`*scope*`?`*filter*

Przykładem może być poniższy adres-zapytanie:

```
ldap://localhost/dc=put,dc=pl??sub?(objectClass=posixAccount)
```

7.2.3 Graficzna przeglądarka GQ

Program GQ domyślnie umożliwia przeglądanie w trybie anonimowym, a więc bez możliwości wprowadzania modyfikacji do serwera. Przejście do trybu edycji wymaga modyfikacji konfiguracji w menu *File/Preferences* i następnie w zakładce *Servers*.



Rys. 7.1: Edytor graficzny GQ

7.2.4 Inne narzędzia

`slapadd`
: Program pozwala na zainicjowanie pustej bazy danych.

`slapcat`
: Wyświetla zawartość bazy danych w formacie LDIF. Program może być wykorzystany do tworzenia kopii zapasowych.

`slapindex`
: Program przebudowuje indeksy bazy danych.

7.3 Prawa dostępu

Prawa dostępu użytkowników do serwera LDAP zapisane są w głównym pliku konfiguracyjnym `/etc/openldap/slapd.conf`. Oto przykładowe reguły:

```
access to attrs=userPassword
       by self write
       by * auth

access to *
       by self write
       by * read
```

Powyższe reguły dają prawo właścicielowi do modyfikacji własnego hasła i pozostałych atrybutów oraz prawo do odczytu wszystkich pozostałych rekordów.

7.4 Integracja z systemem

7.4.1 Moduł nss_ldap

W pliku `/etc/nsswitch.conf(5)` należy zaznaczyć, że dane mają pochodzić z serwera LDAP:

```
passwd: files ldap
```

Biblioteka `libnss_ldap.so` łączy się z serwerem LDAP odczytując swoją konfigurację z pliku `/etc/ldap.conf`. Zawartość pliku wskazuje na serwer LDAP:

```
host         127.0.0.1
base         dc=put,dc=pl
rootbinddn   cn=admin,dc=put,dc=pl
pam_filter   objectClass=posixAccount
pam_password crypt
ssl          no
nss_base_passwd   ou=people,dc=put,dc=pl
nss_base_shadow   ou=people,dc=put,dc=pl
nss_base_group    ou=groups,dc=put,dc=pl
nss_base_hosts    ou=hosts,dc=put,dc=pl
```

Poprawne skonfigurowany system powinien dawać możliwość odczytania informacji o nowych użytkownikach:

```
# getent passwd wojtek
wojtek:x:2123:110:Wojciech Kowalski:/home/wojtek:/bin/bash
```

Ustawienie parametru `rootbinddn` umożliwia zmianę hasła zwykłym poleceniem `passwd` przez administratora systemu. Hasło dostępu do serwera LDAP przechowywane jest w postaci jawnej w pliku /etc/ldap.secret, który powinien być zabezpieczony przed niepowołanym dostępem (prawa 0600).

7.4.2 Moduł pam_ldap

Logowanie na konto użytkownika z serwera LDAP wymaga skonfigurowania modułu PAM. W przypadku pracy interaktywnej zmianie musi ulec więc plik `/etc/pam.d/login`:

```
#%PAM-1.0
auth      required      pam_securetty.so
auth      required      pam_nologin.so
auth      sufficient    pam_ldap.so
auth      required      pam_unix2.so    use_first_pass
auth      required      pam_env.so
auth      required      pam_mail.so
account   sufficient    pam_ldap.so
account   required      pam_unix2.so
password  required      pam_pwcheck.so  nullok
password  sufficient    pam_ldap.so     use_first_pass
password  required      pam_unix2.so    nullok use_first_pass use_authtok
session   required      pam_unix2.so    none
session   required      pam_limits.so
```

Powyższa konfiguracja umożliwia również zmianę hasła użytkownika. Konfigurację warto sprawdzić wykonując następujące testy:

1. Zmiana hasła użytkownika LDAP.
2. Zmiana hasła użytkownika lokalnego (np. `root`).

7.5 Znaki narodowe

Serwery LDAPv3 przechowują wartości atrybutów stosując kodowanie UTF-8. Wprowadzenie łańcuchów tekstowych zawierających znaki spoza podstawowego zestawu ASCII jest możliwe poprzez aplikacje graficzne, lub wsadowo po zakodowaniu zgodnie ze standardem Base64. Można do tego celu wykorzystać program `mimencode(1)`:

```
# printf "Łukasz Żółtkiewicz" | mimencode
xYF1a2FzeiDFu8OzxYJ0a2lld2ljeg==
```

Tak zakodowane wartości atrybutów w formacie LDIF zapisywane są z użyciem podwójnego znaku „:”:

```
dn:             uid=lukasz,ou=people,dc=put,dc=pl
objectClass:    account
objectClass:    posixAccount
cn::            xYF1a2FzeiDFu8OzxYJ0a2lld2ljeg==
...
```

Do konwersji plików tekstowych kodowanych w innym formacie niż UTF-8 najlepiej wykorzystać program `iconv(1)`, np.:

```
# iconv -f ISO8859-2 -t UTF-8 dane.txt > wynik.txt
```

7.6 Bezpieczeństwo — SSL/TLS

Dostęp do serwera LDAP może być realizowany w sposób bezpieczny za pośrednictwem protokołu SSL/TLS. Praca protokołów bezpieczeństwa wymaga dostępu do odpowiednich certyfikatów. Uzyskanie ich wymaga wykonania następujących kroków.

Utworzenie urzędu certyfikującego:

```
# /usr/share/ssl/misc/CA.sh/CA.sh -newca
```

Utworzenie certyfikatu:

```
# openssl req -new -nodes -keyout newreq.pem -out newreq.pem
```

Podpisanie certyfikatu:

```
# /usr/share/ssl/misc/CA.sh/CA.sh -sign
```

Instalacja:

```
# cp demoCA/cacert.pem /etc/openldap/cacert.pem
# mv newcert.pem /etc/openldap/ldap-crt.pem
# mv newreq.pem /etc/openldap/ldap-key.pem
# chmod 600 /etc/openldap/ldap-key.pem
# chown ldap /etc/openldap/ldap-key.pem
```

Konfiguracja serwera (plik /etc/openldap/slapd.conf):

```
TLSCACertificateFile /etc/openldap/cacert.pem
TLSCertificateFile /etc/openldap/ldap-crt.pem
TLSCertificateKeyFile /etc/openldap/ldap-key.pem
```

Konfiguracja klientów (plik /etc/openldap/ldap.conf):

```
tls_cacert /etc/openldap/cacert.pem
```

Konfiguracja modułu `nss_ldap` (plik /etc/ldap.conf):

```
ssl start_tls
```

7.7 Integracja z innymi aplikacjami

7.7.1 Samba

Ze względu na przechowywanie w drzewie katalogowym wielu dodatkowych atrybutów, baza informacji katalogowej powinna zostać uzupełniona dodatkowymi indeksami. Wymaga to po pierwsze modyfikacji odpowiednich zapisów w pliku /etc/openldap/slapd.conf:

```
# Indices to maintain
index objectClass eq
index cn          pres,sub,eq
index sn          pres,sub,eq
## required to support pdb_getsampwnam
index uid         pres,sub,eq
## required to support pdb_getsambapwrid()
```

```
index displayName pres,sub,eq
## uncomment these if you are storing posixAccount and
## posixGroup entries in the directory as well
index uidNumber   eq
index gidNumber   eq
index memberUid   eq
index sambaSID    eq
index sambaPrimaryGroupSID eq
index sambaDomainName eq
index default     sub
```

Następnie należy wymienione indeksy fizycznie utworzyć:

```
# /etc/init.d/ldap stop
# slapindex
# /etc/init.d/ldap start
```

Serwer Samba musi mieć dostęp do danych przechowywanych na serwerze LDAP. Dostęp ten uzyskuje podając hasło, które należy zainicjować wydając komendę:

```
# smbpasswd -w hasło
```

Hasło jest zapisywane w pliku /etc/samba/secrets.tdb.

Zadania

1. Skonfiguruj i uruchom serwer OpenLDAP zaczynając od pustej bazy informacji katalogowej.
2. Dodaj podstawowe rekordy do bazy informacji katalogowej oraz rekordy opisujące kilku użytkowników i kilka grup użytkowników.
3. Przećwicz przeszukiwanie struktury katalogowej. Wykorzystaj różne zakresy przeszukiwania (`base`, `one`, `sub`). Wyświetl konkretne, wybrane atrybuty odnalezionych obiektów. Posłuż się filtrami zawierającymi złożone warunki (koniunkcja, alternatywa, negacja).
4. Dokonaj modyfikacji wybranych atrybutów obiektów.
5. Przećwicz dostęp do serwera LDAP z poziomu przeglądarki internetowej. Wykorzystaj w tym celu przeglądarkę Konqueror pochodzącą ze środowiska graficznego KDE.
6. Przećwicz dostęp do serwera LDAP poprzez klienta graficznego GQ.
7. Ogranicz prawa dostępu do wybranych fragmentów drzewa katalogowego. Ograniczenie powinno dotyczyć wybranych użytkowników. Sprawdź różnice pomiędzy prawami `auth`, `compare` i `search`.
8. Zintegruj usługę LDAP z systemem operacyjnym poprzez udostępnienie użytkowników zdefiniowanych w bazie informacji katalogowej. Użytkownicy powinni mieć możliwość normalnego zalogowania się do systemu.
9. Przećwicz wprowadzanie do usługi katalogowej napisów zawierających polskie litery.
10. Zabezpiecz dostęp do serwera LDAP wykorzystując protokół SSL/TLS.

Skorowidz

domena NIS, 79

filtr XDR, 11
funkcja
    callrpc, 4
    kill, 7
    utimes, 19

grupy sieciowe, 57

Konqueror, 59

mapa systemu NIS, 79

plik konfiguracyjny
    auto.home, 81
    auto.master, 59, 81
    exports, 55, 56
    group, 81
    hosts, 81
    netgroup, 57, 81
    nsswitch.conf, 70, 82, 92
    passwd, 81
    rpc, 81
    securenets, 84
    services, 81
    shadow, 81
    slapd.conf, 87
polecenie
    chfn, 83
    chsh, 83
    domainname, 79
    hostname, 79
    iconv, 93
    ldapadd, 89
    ldapdelete, 90
    ldapmodify, 90
    ldapsearch, 89
    make, 80
    mimencode, 93
    nisdomainname, 79
    passwd, 83
    rpc.rusersd, 6
    rpcgen, 7, 14
    rpcinfo, 5
    slappasswd, 87
    smbclient, 64
    smbmount, 65
    wbinfo, 71
    ypbind, 82
    ypcat, 81
    ypchfn, 83
    ypchsh, 83
    ypdomainname, 79
    ypinit, 83
    ypmatch, 82
    yppasswd, 83
    ypwhich, 82
potok XDR, 11
protokół transportowy, 8

usługa
    rusers, 5

WebNFS, 59

Bibliografia

[ECBK03] R. Eckstein, D. Collier-Brown, and P. Kelly. *Using Samba.* O'Reilly, 2nd edition, February 2003.

[Fre01] Free Software Foundation. *The GNU C Library Reference Manual*, 2001.

[GD95] M. Gabassi and B. Dupouy. *Przetwarzanie rozproszone w systemie UNIX.* Lupus, W-wa, 1995.

[Gra98] J. S. Gray. *Arkana: Komunikacja między procesami w Unixie.* READ ME, 1998.

[ION01] IONA Technologies. *CORBA/C++ Programming with ORBacus — Student Workbook*, September 2001. Dokument dostępny pod adresem http://www.orbacus.com, sekcja *Support*, odnośnik *Training.*

[Mic87] Sun Microsystems. XDR: External Data Representation standard. RFC 1014, June 1987.

[Mic88] Sun Microsystems. RPC: Remote Procedure Call Protocol specification. RFC 1050 (Historic), April 1988. Obsoleted by RFC 1057.