Programowanie sieciowe

Wojciech Palacz

semestr letni 2023/24

(robocza wersja slajdów z 2024-03-01)

Cele wykładu

Po zaliczeniu tego przedmiotu powinni Państwo:

- znać zaimplementowane w jądrach systemów operacyjnych i w pomocniczych bibliotekach mechanizmy, dzięki którym procesy mogą komunikować się poprzez sieć
- umieć do postawionego przed Wami problemu dobrać odpowiedni mechanizm komunikacji sieciowej

Nie wymaga się od Państwa znania na pamięć wszystkich nazw funkcji udostępnianych przez różne sieciowe API. Musicie natomiast rozumieć sposób działania tych API oraz wiedzieć co przy ich pomocy da się, a czego nie da się zaimplementować.

Wzorce komunikacyjne

Przy analizowaniu systemu rozproszonego trzeba ustalić kto z kim się komunikuje, jakie dane sobie przesyłają i czy korzystają z pomocy pośredników. Wzorców wymiany danych jest wiele. Kilka przykładów:

- przeglądarka pobierająca zasób z witryny WWW (klient-serwer)
- wieloosobowa gra-strzelanka działająca w sieci lokalnej (peer-to-peer)
- list elektroniczny od prof. Nowaka do prof. Kowalskiego (store and forward)
- zawiadomienie o zajściu zdarzenia, np. wpłacie określonej kwoty na określone konto, zaadresowane „do wszystkich zainteresowanych” (publisher-subscribers)

Protokoły komunikacyjne

Protokół to zbiór zasad postępowania (czy to na balu u królowej, czy też podczas łączenia się z witryną WWW i proszenia o udostępnienie pewnego zasobu). Jeśli chcemy zaimpl. grę-strzelankę, to wcześniej musimy zdefiniować protokół wymiany informacji pomiędzy graczami.

Na wykładzie omawiane jest tworzenie protokołów aplikacyjnych, czyli definiujących format i znaczenie komunikatów przesyłanych pomiędzy procesami. Nie będziemy próbowali określać, jak te komunikaty mają być przesyłane przez łącza telekomunikacyjne — za to zadanie będą odpowiadać protokoły transportowe z niższej warstwy modelu ISO/OSI. Twórca aplikacji nie musi znać technicznych szczegółów ich działania aby móc z nich korzystać.

Protokoły komunikacyjne

Aby móc użyć protokołu transportowego trzeba wiedzieć:

- co jest odpowiednikiem adresu pocztowego lub numeru telefonu?
- czy dane przesyłane są w datagramach, czy strumieniowo?
- co się dzieje, gdy odbiorca nie słucha?
- co, gdy jakieś dane zaginą albo zostaną zniekształcone podczas transmisji?
- jak sygnalizować chęć zakończenia wymiany danych?

Procesy komunikujące się za pomocą datagramów przypominają ludzi wysyłających do siebie listy (zwykłe, a nie elektroniczne).

Komunikacja strumieniowa przypomina zaś rozmowę telefoniczną.

Obsługa sieci we współczesnych systemach operacyjnych

TCP i UDP to najpopularniejsze protokoły transportowe. Moduły obsługujące je oraz protokoły znajdujące się „pod nimi" są częścią jądra systemu operacyjnego, a korzystanie z ich usług jest możliwe za pośrednictwem tzw. gniazdek sieciowych (ang. network sockets).

Gniazdka pojawiły się na początku lat 80 w systemie BSD Unix. Można o nich myśleć jako o rozszerzeniu tradycyjnych deskryptorów plików. Rozpowszechniły się w innych systemach operacyjnych, a następnie z pewnymi modyfikacjami zostały włączone do standardu POSIX.

Z punktu widzenia twórcy aplikacji komunikacja przez TCP i UDP to ten sam poziom abstrakcji co zapisywanie danych w plikach. I tu, i tu mamy do czynienia z ciągami bajtów. To, jak te ciągi interpretować, zależy od aplikacji.

Uniksowe deskryptory plików

Standard POSIX specyfikuje cztery podstawowe funkcje:

```
int open(const char * path, int oflag, ...);

ssize_t read(int fildes, void * buf, size_t nbyte);

ssize_t write(int fildes, const void * buf, size_t nbyte);

int close(int fildes);
```

Proszę zwrócić uwagę na sposób raportowania błędów oraz innych nietypowych sytuacji.

Deskryptory nie-plików

Systemy uniksowe stosują się do zasady „wszystko jest plikiem”.
Deskryptor może odnosić się do:

- zwykłego pliku na dysku
- urządzenia USB (myszki, klawiatury, ...)
- portu szeregowego, do którego jest podłączony modem
- potoku łączącego dwa spokrewnione procesy
- itd.

Gniazdka sieciowe bez trudu wpisały się w ten paradygmat.

W przypadku nie-plików jądro systemu może oferować dodatkowe operacje, które nie miałyby sensu dla zwykłych plików. Np. prędkość transmisji portu szeregowego można zmienić za pomocą `ioctl()`, a potoki tworzy się nie przy pomocy `open()`, lecz `pipe()`.

Zarys API gniazdek

Gniazdko pozwala przesyłać dane za pomocą określonego protokołu transportowego. Tworzy się je wywołując `socket()` z argumentami określającymi typ gniazdka (TCP/IPv4, UDP/IPv4, TCP/IPv6, ...).

Gniazdko można połączyć ze wskazanym miejscem docelowym przy pomocy `connect()`, a następnie czytać/pisać z niego jak z każdego innego deskryptora.

Alternatywnie, gniazdko można wprowadzić w tryb nasłuchiwania przy pomocy `bind()` oraz `listen()`. Potem wywołując `accept()` można odbierać przychodzące połączenia.

Na gniazdku w stanie „nawiązano połączenie" można wywoływać funkcje `read()` i `write()`, tak jak gdyby był to zwykły deskryptor pliku.

Zarys API gniazdek

`read()` i `write()` mają swoje rozszerzone wersje: `recv()` i `send()`. Te dwie funkcje mają dodatkowy parametr, pozwalający przekazać do jądra tzw. flagi, czyli liczbę typu `int` traktowaną jako zbiór bitów. Ustawiając wybrane bity-flagi na 1 można zażądać wykonania operacji we-wy w nietypowy sposób.

Dostępne są stałe odpowiadające flagom, np. `MSG_DONTWAIT` (zamiast zawieszania się zwróć -1), `MSG_PEEK` (podglądanie danych bez ich odczytu), itd.

Dla gniazdek UDP działających w trybie bezpołączeniowym przeznaczone są `recvfrom()` i `sendto()`.

API gniazdek: dziedziny

Gniazdka zaprojektowano tak, aby ich API było uniwersalne, i aby jądro systemu mogło równolegle obsługiwać wiele nie związanych ze sobą protokołów sieciowych. Pozwala to np. stworzyć serwer WWW obsługujący zapytania przychodzące i po IPv4, i po IPv6, albo bramkę pośredniczącą w wymianie listów pomiędzy lokalną siecią Novell a Internetem.

Każde stworzone gniazdko należy do pewnej dziedziny (czy też domeny) komunikacyjnej i potrafi obsługiwać połączenia w obrębie tej dziedziny. Proces może mieć równocześnie otwartych kilka gniazdek należących do różnych dziedzin.

Oprócz nazwy „dziedzina/domena” używa się również nazw „rodzina protokołów” oraz „rodzina adresowa” (ang. address family).

API gniazdek: dziedziny

POSIX definiuje następujące stałe używane do oznaczania dziedzin:

- `AF_INET` — IPv4
- `AF_INET6` — IPv6
- `AF_UNIX` — gniazdka lokalne, widziane jako pseudopliki

Linux obsługuje ponad 30 dodatkowych rodzin protokołów. Są to m.in.:

- `AF_IPX` — protokół sieci rozległych Novella
- `AF_APPLETALK`
- `AF_BLUETOOTH`

W ramach danej rodziny zazwyczaj istnieje jeden protokół sieciowy i kilka korzystających z niego protokołów transportowych.

API gniazdek: typy gniazdek

Założono, że każdy z protokołów obsługiwanych przez gniazdka można zaliczyć do jednego z kilku ogólnych typów. POSIX definiuje:

- `SOCK_STREAM` — godny zaufania strumień bajtów (np. TCP)
- `SOCK_SEQPACKET` — godne zaufania datagramy
- `SOCK_DGRAM` — zawodne datagramy (np. UDP)
- `SOCK_RAW` — bezpośredni dostęp do niższych warstw (tylko do zastosowań specjalnych)

Niektóre systemy operacyjne definiują dodatkowe typy:

- `SOCK_RDM` — datagramy z potwierdzeniem odbioru i retransmisją w razie konieczności, ale nie gwarantujące zachowania kolejności
- `SOCK_CONN_DGRAM` — datagramy z zachowaniem kolejności, ale bez gwarancji że wszystkie dotrą do odbiorcy

API gniazdek: funkcja `socket`

Nowe gniazdka tworzone są za pomocą funkcji `socket`:

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

Jako trzeci argument przeważnie podaje się zero, jądro systemu wybiera wtedy domyślny protokół (np. w dziedzinie INET standardowym protokołem strumieniowym jest TCP).

Wartość różną od zera trzeba podać tylko wtedy, gdy w danej rodzinie są dwa protokoły danego typu, i chcemy użyć tego drugiego:

```
int fd = socket(AF_INET, SOCK_STREAM, IPPROTO_SCTP);
```

Powyższa instrukcja stworzy gniazdko korzystające z SCTP/IPv4.

API gniazdek: adresy rodziny INET

```
typedef uint32_t in_addr_t;
typedef uint16_t in_port_t;

struct in_addr {
    in_addr_t s_addr;   // w sieciowej kolejności bajtów
};

struct sockaddr_in {
    sa_family_t sin_family;
    in_port_t sin_port;   // w sieciowej kolejności bajtów
    struct in_addr sin_addr;
    ..... // struktura może zawierać dodatkowe pola
};
```

API gniazdek: adresy rodziny INET

Ręczne wypełnianie struktury (adres 1.2.3.4, port 80):

```
sockaddr_in adres;
memset(&adres, 0, sizeof(adres));
adres.sin_family = AF_INET;
adres.sin_port = htons(80);
adres.sin_addr.s_addr = htonl(
    ((1 * 256 + 2) * 256 + 3) * 256 + 4
);
```

Jeśli adres IP jest dany jako łańcuch można użyć funkcji bibliotecznych:

```
adres.sin_addr.s_addr = inet_addr("1.2.3.4");
```

albo

```
inet_pton(AF_INET, "1.2.3.4", &adres.sin_addr);
```

API gniazdek: adresy rodziny INET6

Wszytkie pola z wyjątkiem rodziny w sieciowej kolejności bajtów.

```
typedef uint16_t in_port_t;

struct in6_addr {
    unsigned char s6_addr[16];   // 16 bajtów = 128 bitów
};

struct sockaddr_in6 {
    sa_family_t sin6_family;
    ..... // struktura może zawierać dodatkowe pola
    in_port_t sin6_port;
    uint32_t sin6_flowinfo;
    struct in6_addr sin6_addr;
    uint32_t sin6_scope_id;
};
```

API gniazdek: inne struktury adresowe

Każda rodzina ma swoją własną dedykowaną strukturę. Pierwsze pole zawsze jest typu `sa_family_t` (jakiś typ całkowitoliczbowy bez znaku).

- `struct sockaddr_un` — pola `sun_family` i `sun_path`
- `struct sockaddr_atalk` — pola `sat_family`, `sat_port` i `sat_addr` (podstruktura z polami `s_net` i `s_node`)
- itd.

POSIX dodatkowo przewiduje dwie struktury specjalnego przeznaczenia, używane przy przekazywaniu/zwracaniu adresów do/z jądra:

- `struct sockaddr` — pole `sa_family`
- `struct sockaddr_storage` — pole `ss_family` i dodatkowe wypełnienie, dzięki czemu rozmiar tej struktury jest co najmniej taki jak największej z pozostałych struktur

API gniazdek: funkcja connect

```
int connect(int socket, const struct sockaddr * address,
        socklen_t address_len);
```

API musi być uniwersalne, więc typem wskaźnika pokazującego na strukturę z adresem jest `struct sockaddr`. Rzeczywisty rozmiar struktury jest przekazywany jako dodatkowy parametr.

```
int fd = socket(AF_INET, SOCK_STREAM, 0);
struct sockaddr_in adres;
..... // wypełnienie struktury adresem serwera
connect(fd, (struct sockaddr *) &adres, sizeof(adres));
```

API gniazdek: funkcje getsockname i getpeername

```
int getsockname(int socket, struct sockaddr * address,
        socklen_t * address_len);
int getpeername(int socket, struct sockaddr * address,
        socklen_t * address_len);
```

Jeśli program używa gniazdek z kilku różnych domen i nie wiadomo z którym mamy do czynienia, to adres lokalnego/zdalnego końca połączenia najwygodniej pobrać do `struct sockaddr_storage`, na pewno się tam w całości zmieści.

```
struct sockaddr_storage adres;
socklen_t dlugosc = sizeof(adres);
getsockname(fd, (struct sockaddr *) &adres, &dlugosc);
if (adres.ss_family == AF_APPLETALK) {
    .....
} else if (adres.ss_family == AF_INET) .....
```

API gniazdek: funkcja `bind`

```
int bind(int socket, const struct sockaddr * address,
        socklen_t address_len);
```

Ustala adres lokalnego końca gniazdka. Klienci nie muszą z niej korzystać, `connect()` potrafi automatycznie wybrać jakiś wolny lokalny adres. Serwery zazwyczaj specyfikują tylko numer portu, adres IP pozostawiając nieokreślony (wtedy odbiera połączenia na wszystkich IP komputera):

```
sockaddr_in adres;
memset(&adres, 0, sizeof(adres));
adres.sin_family = AF_INET;
adres.sin_port = htons(8080);
adres.sin_addr.s_addr = htonl(INADDR_ANY);
```

Porty 0-1023 są zarezerwowane dla procesów z prawami administratora.

API gniazdek: funkcje `listen` i `accept`

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

Oznacza gniazdko jako nasłuchujące i ustala długość kolejki połączeń oczekujących.

```
int accept(int socket, struct sockaddr * address,
        socklen_t * address_len);
```

Czeka na połączenie, tworzy nowe gniazdko reprezentujące to połączenie i zwraca jego deskryptor. Jeśli drugi argument jest różny od NULL, to zwraca również adres klienta, który z nami się połączył.

API gniazdek: zarys kodu najprostszego serwera

```
int serv_fd = socket(AF_INET, SOCK_STREAM, 0);
bind(serv_fd, .....);
listen(serv_fd, .....);
while (1) {
    int fd = accept(serv_fd, NULL, 0);
    ..... // pisz i/lub czytaj z fd
    close(fd);
}
```

API gniazdek: funkcja `shutdown`

Jeśli gniazdko reprezentuje dwukierunkowe połączenie, to można zamknąć je tylko w jednym kierunku, drugi pozostawiając otwarty.

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

Stałe podawane jako drugi argument:

- `SHUT_RD`
- `SHUT_WR`
- `SHUT_RDWR` — oba kierunki naraz, tak jak `close()`

Najprzydatniejsze jest `SHUT_WR`, można w ten sposób zasygnalizować koniec wysyłanych przez nas danych a potem jeszcze odebrać finalną odpowiedź od procesu na drugim końcu gniazdka.

Gniazdka bezpołączeniowe

Gniazdka UDP mogą wysyłać i odbierać datagramy w tzw. trybie bezpołączeniowym, w którym gniazdko nie ma ustalonego adresu zdalnego. Każdy transmitowany datagram może mieć inny adres zdalny.

```
ssize_t sendto(int socket,
    const void * message, size_t length, int flags,
    const struct sockaddr * dest_addr, socklen_t addr_len);


ssize_t recvfrom(int socket,
    void * buffer, size_t length, int flags,
    struct sockaddr * src_addr, socklen_t * addr_len);
```

Flagi najczęściej ustawia się na zero.

Najprostszy bezpołączeniowy serwer wywołuje `socket()`, `bind()`, potem w nieskończonej pętli `recvfrom()` i `sendto()`. Adres, na który należy odesłać odpowiedź jest znany, bo zwraca go `recvfrom()`.

Nieblokujące deskryptory

- Operacje odczytu, które zawiesiłyby się z powodu braku gotowych danych zamiast tego zwracają -1 i ustawiają w `errno` kod błędu EAGAIN lub EWOULDBLOCK (w użyciu są dwa różne kody z uwagi na historyczne różnice pomiędzy systemami operacyjnymi).
- Tak samo operacje zapisu, które zawiesiłyby się z powodu braku miejsca w buforach jądra.
- Można ich używać razem z select/poll/itd. — gdy dany deskryptor stanie się gotowy, wywołuj w pętli read (czy też write) aż zamiast danych zwróci EAGAIN (czy też EWOULDBLOCK).

```
int flags = fcntl(fd, F_GETFL);
flags |= O_NONBLOCK;
fcntl(fd, F_SETFL, flags);
```

Gniazdka z timeoutem

- Operacje odczytu/zapisu blokują się, ale tylko na czas nie dłuższy niż ustawiony timeout.
- Po przekroczeniu timeoutu zwracane jest -1 z kodem EAGAIN lub EWOULDBLOCK.

```
struct timeval timeout = ...;
setsockopt(sock_fd, SOL_SOCKET,
    SO_RCVTIMEO, &timeout, sizeof(timeout));
setsockopt(sock_fd, SOL_SOCKET,
    SO_SNDTIMEO, &timeout, sizeof(timeout));
```

Inne interesujące opcje gniazdek

Na poziomie SOL_SOCKET:

- możliwość wysyłania datagramów na adres broadcastowy

```
int i = 1;   // interpr. jako wartość logiczna
setsockopt(s, SOL_SOCKET, SO_BROADCAST, &i, sizeof(i));
```

- okresowe wysyłanie pustych pakietów podtrzymujących połączenie

```
int i = 1;
setsockopt(s, SOL_SOCKET, SO_KEEPALIVE, &i, sizeof(i));
```

Na poziomie IPPROTO_TCP:

- natychmiast wysyłaj w świat nawet pojedyncze bajty (tzn. nie używaj algorytmu Nagle'a)

```
int i = 1;
setsockopt(s, IPPROTO_TCP, TCP_NODELAY, &i, sizeof(i));
```

Gniazdka w Javie

Klasa `java.net.Socket` opakowuje klienckie gniazdko TCP.
Wybrane konstruktory i metody:

- `Socket()`
- `Socket(InetAddress address, int port)`
- `Socket(InetAddress address, int port,`
  `InetAddress localAddr, int localPort)`
- `void bind(SocketAddress bindpoint)`
- `void connect(SocketAddress endpoint)`
- `InputStream getInputStream()`
- `OutputStream getOutputStream()`
- `void close()`
- `void shutdownInput()`
- `void shutdownOutput()`

Gniazdka w Javie

Klasa `java.net.ServerSocket` to serwerowe gniazdko TCP.
Wybrane konstruktory i metody:

- `ServerSocket()`
- `ServerSocket(int port)`
- `ServerSocket(int port, int backlog,`
  `InetAddress bindAddr)`
- `void bind(SocketAddress bindpoint, int backlog)`
- `Socket accept()`
- `void close()`

Gniazdka w Javie

Klasa `java.net.DatagramSocket`:

- `DatagramSocket()`
- `DatagramSocket(int port, InetAddress bindAddr)`
- `void bind(SocketAddress addr)`
- `void receive(DatagramPacket p)`
- `void send(DatagramPacket p)`
- `void close()`

Obiekty typu `DatagramPacket` zawierają tablicę bajtów oraz adres, na który wysłać / z którego przyszedł dany datagram.

Nano-wprowadzenie do Pythona 3

Wydanie polecenia `python3` bez żadnych argumentów uruchamia interpreter działający w trybie REPL.

```
>>> 2 + 2
4
>>> import math
>>> math.cos(0)
1.0
>>> x = 3 * 7
>>> x
21
>>> x = math.sin(math.pi / 4)
>>> x
0.7071067811865475

>>> exit()
```

Nano-wprowadzenie do Pythona 3

Łańcuchy w Pythonie 3 są unikodowe. Literały łańcuchowe można ujmować w apostrofy albo w cudzysłowy, bez różnicy.

```
>>> s = 'Witaj, świecie!'
>>> s
'Witaj, świecie!'
>>> s[0]
'W'
>>> len(s)
15
>>> s[15]
IndexError: string index out of range
>>> s[2:5]
'taj'
>>> type(s)
<class 'str'>
```

Nano-wprowadzenie do Pythona 3

Oprócz łańcuchów są ciągi bajtów. Literały tego typu można specyfikować jako łańcuchy ASCII poprzedzone prefiksem 'b'.

```
>>> b = b'Witaj, świecie!'
SyntaxError: bytes can only contain ASCII literal characters.
>>> b = b'Hello, world!'
>>> b
b'Hello, world!'
>>> b[0]
72
>>> b[1:3]
b'el'
>>> type(b)
<class 'bytes'>
```

Gniazdka w Pythonie 3

```
>>> import socket
>>> s = socket.socket()
>>> s
<socket.socket object, fd=3, family=2, type=1, proto=0>
>>> socket.AF_INET
2
>>> socket.SOCK_STREAM
1
>>> s.getpeername()
socket.error: [Errno 107] Transport endpoint is not connected
>>> s.getsockname()
('0.0.0.0', 0)

>>> help(s)
```

Gniazdka w Pythonie 3

```
>>> s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
>>> a = ('spk-ssh.if.uj.edu.pl', 22)
>>> s.connect(a)
>>> s.getpeername()
('149.156.43.64', 22)
>>> s.getsockname()
('1.2.3.4', 44594)
>>> s.recv(1024)
b'SSH-2.0-OpenSSH_7.6p1 Ubuntu-4ubuntu0.3\r\n'
>>> s.close()
>>> s
<socket.socket [closed]  object, fd=-1,
                    family=2, type=1, proto=0>
```

Gniazdka w Pythonie 3

```
>>> s = socket.socket()
>>> s.connect( ('smtp.gmail.com', 25) )
>>> s.recv(1024)
b'220 smtp.gmail.com ESMTP u29xxx.84 - gsmtp\r\n'
>>> s.send(b'QUIT\r\n')
6
>>> s.recv(1024)
b'221 2.0.0 closing connection u29xxx.84 - gsmtp\r\n'
>>> s.recv(1024)
b''
>>> s.close()
```

Podsumowanie API gniazdek

API gniazdek jest jednym z API udostępnianych przez jądra POSIX-owych systemów operacyjnych. Mamy do niego bezpośredni dostęp z poziomu języków C i C++.

Języki obiektowe często ukrywają deskryptory gniazdek wewnątrz obiektów, aby ułatwić życie programistom przyzwyczajonym do obiektowego stylu programowania, automatycznego odśmiecania, itd.

API gniazdek definiuje sposób, w jaki proces może przesyłać ciągi bajtów innemu procesowi, zazwyczaj działającemu na innym komputerze. To wszystko — w specyfikacji gniazdek nie znajdziemy nic ani na temat kart sieciowych, ani na temat znaczenia przesyłanych bajtów. Te rzeczy opisane są w innych specyfikacjach i standardach.

Model ISO/OSI

Rozległe sieci telekomunikacyjne (np. Internet) to bardzo skomplikowane systemy. Zrozumienie i analiza ich działania są możliwe dopiero po rozbiciu ich na komponenty działające na różnych poziomach abstrakcji.

Model ISO/OSI proponuje podział na siedem warstw. Każda z nich kontaktuje się tylko z warstwami sąsiednimi (to ułatwia analizę oraz implementowanie modułów odpowiedzialnych za poszczególne warstwy).

Model ISO/OSI jest najczęściej wykorzystywany jako referencyjny model koncepcyjny mający organizować nasze myślenie. Używane w praktyce rozwiązania przeważnie nie są w 100% z nim zgodne. W szczególności tzw. model internetowy ma tylko cztery warstwy.

Warstwy modelu ISO/OSI

Od najniższej do najwyższej:

1. fizyczna
2. łącza danych
3. sieciowa
4. transportowa
5. sesji
6. prezentacji
7. aplikacji

Człowiek-użytkownik jest w „warstwie 8”.

Rzeczywiste standardy: Ethernet to warstwy 1 i 2, IP warstwa 3, TCP i UDP 4 z elementami 5, poczta, WWW, SSH itd. to warstwa 7.

Adresowanie w poszczególnych warstwach

Ethernet: adresy 48-bitowe, przydzielane w fabryce, producenci sprzętu zapewniają ich globalną unikalność.

Wi-Fi: ten standard zaprojektowano aby był kompatybilny z Ethernetem, wspólna przestrzeń adresowa.

IPv4: adresy 32-bitowe, przydział zasadniczo koordynowany przez operatorów sieci szkieletowych aby zapewnić globalną unikalność.

IPv6: adresy 128-bitowe, reszta jak wyżej.

TCP: 16-bitowe numery portów (istnieją tzw. „dobrze znane numery portów", przydziela je IANA czyli Internet Assigned Numbers Authority).

UDP: jak wyżej, ale odrębna przestrzeń numerów.

Poczta: adresy postaci `jan.kowalski@example.org`.

World Wide Web: URL-e, czyli `http://example.org/page.html` itp.

Serwer dodający liczby

Przykład dydaktyczny: projektowanie protokołu z warstwy aplikacyjnej.

Rozważmy problem implementowania pary klient-serwer, w której klient wysyła ciąg liczb, a serwer zwraca ich sumę. Zanim zaczniemy pisać kod trzeba wcześniej ustalić w jaki sposób klient i serwer będą się ze sobą komunikować, czyli trzeba zdefiniować protokół warstwy aplikacyjnej.

Definicja musi być precyzyjna, tak aby na jej podstawie dało się jednoznacznie stwierdzić, czy dany ciąg bajtów jest poprawnym komunikatem, czy też nie. Co więc w takiej specyfikacji protokołu trzeba koniecznie opisać?

Lista punktów do rozważenia — patrz następny slajd.

Serwer dodający liczby

- Jakie liczby: naturalne, całkowite, inne?
- Protokół binarny czy tekstowy?
- Jeśli binarny, to ile bitów na liczbę i w jakim formacie?
- Jeśli tekstowy ASCII, to czym separujemy sąsiadujące ze sobą liczby?
- W niższej warstwie TCP czy UDP? A może dopuszczamy oba?
- Czy w ramach połączenia klient-serwer można przesłać kilka ciągów?
- Jeśli tak, to jak są one od siebie odseparowane?
- Jak serwer ma reagować na niepoprawne komunikaty?
- Jak reagować na przepełnienia arytmetyczne?

Walidacja danych wejściowych

Przy pisaniu aplikacji sieciowych obowiązuje zasada ograniczonego zaufania. Odczytywane z gniazdek ciągi bajtów nie zawsze będą zgodne ze specyfikacją protokołu.

Przyjmijmy, że zdecydowaliśmy się na tekstowy protokół datagramowy. Zapytanie powinno zawierać jeden lub więcej ciągów cyfr ASCII odseparowanych od siebie pojedynczymi spacjami.

Przykłady poprawnych datagramów-zapytań:

50 32 50 (czyli 2␣2)

53 55 53 32 50 48 32 52 56 (czyli 575␣20␣48)

53 48 48 57 (czyli 5009)

Przykłady zapytań niepoprawnych:

49 48 32 45 53 (czyli 10␣-5)

32 50 32 50 (czyli ␣2␣2)

55 50 32 32 50 56 (czyli 72␣␣28)

Automaty

Do walidacji zapytań można użyć automatu. To pojęcie z teorii informatyki, maszyna obliczeniowa potrafiąca mniej niż maszyna Turinga, ale do wielu zastosowań całkowicie wystarczająca.

Deterministyczny automat skończony (albo „skończenie stanowy”) definiuje się w oparciu o:

- zbiór symboli, z których składają się ciągi dostarczane na wejście automatu
- zbiór stanów, w których automat może się znajdować (jeden z nich jest wyróżniony jako stan początkowy)
- funkcję przejścia, określającą przejścia między stanami pod wpływem przetwarzanych symboli

Automat walidujący ciąg liczb

Podstawowym typem automatu jest automat akceptujący, mający wyróżniony zbiór stanów akceptujących. Jeśli po zakończeniu przetwarzania ciągu wejściowego automat jest w jednym z tych stanów, to ciąg był poprawny.

Zaprojektujmy taki automat dla naszych datagramów-zapytań.

Użyjemy 256 symboli wejściowych odpowiadających wszystkim możliwym wartościom jednego bajtu.

Najwygodniejszym sposobem reprezentowania tych symboli jest użycie liczb z przedziału $[0, 255]$, w ten sposób bajty odczytywane z datagramu można przekazywać wprost na wejście automatu. Liczby-symbole od 48 do 57 odpowiadają cyfrom ASCII, a 32 – spacji.

Automat walidujący ciąg liczb

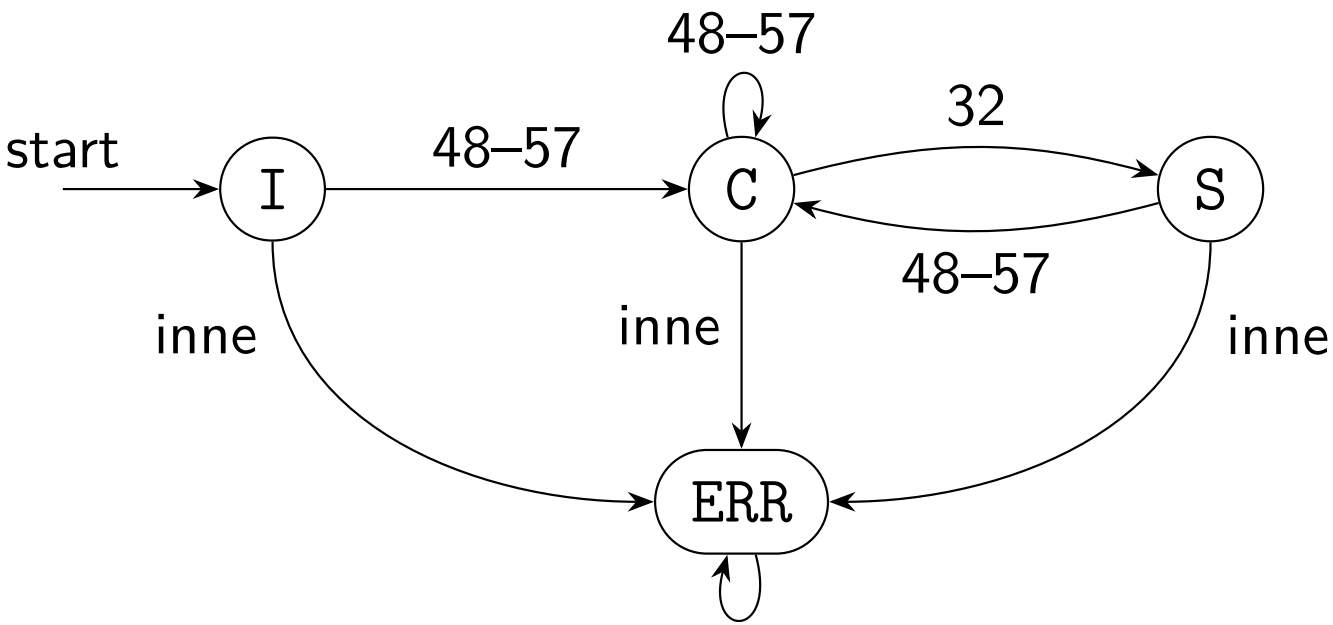
Automat będzie miał 4 stany:

- I stan początkowy, automat nie przetworzył jeszcze żadnego symbolu
- C ostatnio przetworzonym symbolem była cyfra
- S ostatnio przetworzonym symbolem była spacja
- ERR poprzednio przetworzone symbole nie układają się w poprawny ciąg

Jedynym stanem akceptującym jest C.

Automat walidujący ciąg liczb

Funkcję przejść można rozpisać w formie tabeli, ale w tym przypadku czytelniej będzie rozrysować ją jako graf, czyli jako diagram stanów. Wierzchołki to stany, krawędzie to możliwe przejścia, etykiety krawędzi wskazują pod wpływem jakich symboli dane przejście jest wykonywane.



Automaty z akcjami

Serwer, który najpierw waliduje datagram, a dopiero potem przetwarza dane w nim zawarte, dwa razy przegląda te same dane wejściowe. Jeśli jest to możliwe, to warto wykonywać walidację i przetwarzanie w jednym przebiegu. Problem dodawania ciągów liczb na taką optymalizację pozwala.

Zamiast automatu akceptującego użyjemy automatu z akcjami. Akcja jest związana z określonym przejściem pomiędzy stanami i opisuje czynności, które należy wykonać gdy automat takie przejście robi.

Pojęcie „stan automatu z akcjami” często oznacza stan złożonej struktury danych. Tylko jedna wartość w strukturze odpowiada stanowi w ścisłym sensie, pozostałe są wykorzystywane przez akcje.
Jeśli automat implementujemy jako obiekt, to naturalnym podejściem jest utożsamienie stanu automatu ze stanem pól tego obiektu.

Automat obliczający sumę liczb

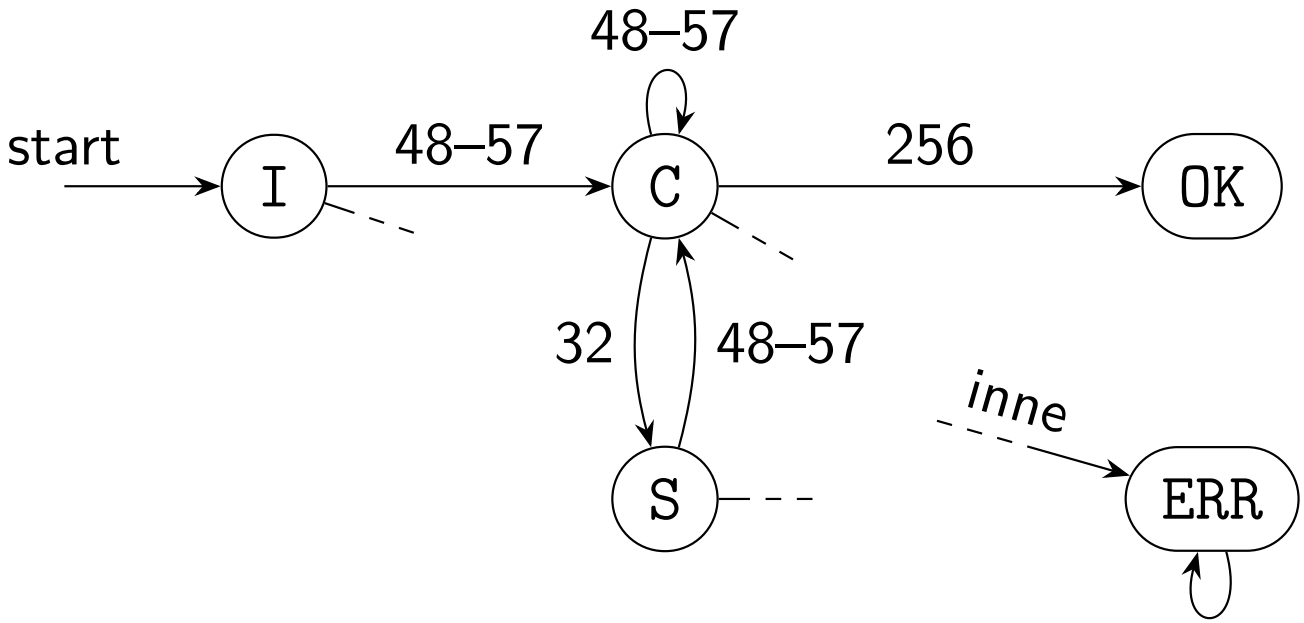
Akcja zwracająca klientowi obliczoną sumę powinna być wykonana dopiero po przetworzeniu całego datagramu. Wymusza to wprowadzenie dodatkowego symbolu oznaczającego dotarcie do końca datagramu.

W użyciu są już liczby-symbole od 0 do 255, niech więc symbolem oznaczającym koniec danych będzie liczba 256.

```
datagram = gniazdko_udp.receive()
au = new Automat()
for bajt in datagram:
    au.consume(bajt)
au.consume(256)
```

Automat obliczający sumę liczb

Pojawia się nowy stan oznaczony OK. Znaczenie: ostatnio przetworzonym symbolem był „koniec datagramu”, a automat nie natknął się wcześniej na żadne błędy.



Automat obliczający sumę liczb

Stan automatu jest wyrażony strukturą składającą się z trzech pól:

- `stan` – zawiera jedną z pięciu wartości odpowiadających etykietom wierzchołków w grafie przejść, automatycznie uaktualniane podczas wykonywania przejścia
- `suma` – pole całkowitoliczbowe, suma przetworzonych do tej pory liczb
- `liczba` – też pole całkowitoliczbowe, używane jako zmienna robocza podczas konwersji ciągu cyfr ASCII na liczbę

Początkowy stan automatu to (`I`, `0`, `0`).

Automat obliczający sumę liczb

Akcje wykonywane przez automat wyglądają tak:

`I` $\rightarrow$ `C`, `C` $\rightarrow$ `C`, `S` $\rightarrow$ `C`

```
liczba = 10 * liczba + (przetwarzany_symbol - 48)
```

`C` $\rightarrow$ `S`

```
suma = suma + liczba
liczba = 0
```

`C` $\rightarrow$ `OK`

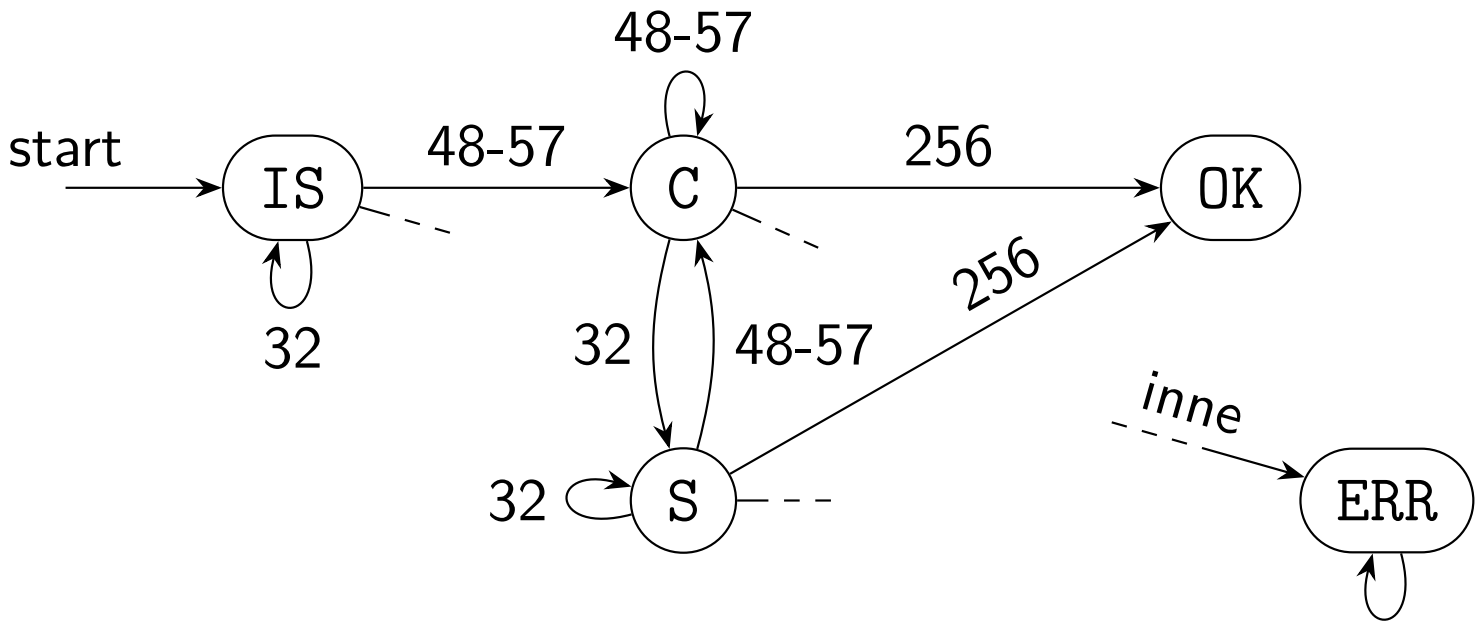
```
suma = suma + liczba
gniazdko_udp.send(suma)
```

`I` $\rightarrow$ `ERR`, `C` $\rightarrow$ `ERR`, `S` $\rightarrow$ `ERR`

```
gniazdko_udp.send("ERROR")
```

Z przejściem `ERR` $\rightarrow$ `ERR` nie jest związana żadna akcja.

Inna specyfikacja protokołu, inny automat



Powyższy automat został zaprojektowany dla trochę innej, nieco więcej dopuszczającej wersji protokołu dodawania liczb.

Pytanie: jakie datagramy, odrzucane przez poprzedni automat, będą przez ten uznawane za poprawne? Czy i jakie akcje należy związać z krawędziami, które w tym grafie dodano?

Automaty i przetwarzanie strumieni danych

Podejście oparte o automaty jest szczególnie wygodne w przypadku protokołów korzystających z transportu strumieniowego i nie mających ograniczenia na długość przesyłanych komunikatów (np. sumator TCP, któremu można kazać dodać do siebie miliard jedynek).

Wczytywanie i przetwarzanie tego mającego 2 GB zapytania musi być robione porcjami o rozsądnym rozmiarze. Automat to umożliwia: wczytaj porcję danych do bufora, przekaż bajt po bajcie automatowi, i ponów.

Automat pamięta swój własny stan, co pozwala serwerowi nie czekać bezproduktywnie na nadejście następnej porcji danych od klienta, lecz zająć się innymi rzeczami, a gdy potem te dane się pojawią wznowić przetwarzanie w dokładnie tym punkcie, w którym je zawieszono.

Specyfikacje protokołów

Specyfikacje większości protokołów internetowych są dostępne na witrynie https://www.rfc-editor.org/ jako tzw. dokumenty RFC (ang. *request for comments*).

Warto zapoznać się z reprezentatywną próbką RFC i wzorować na ich strukturze podczas pisania własnych specyfikacji protokołów.

Protokoły z warstw poniżej gniazdek:

- UDP (RFC 768 / STD 6 — tylko trzy strony)
- IPv4 (RFC 791 z późniejszymi uaktualnieniami, część STD 5)
- TCP (oryg. wersja spec. RFC 793, aktualna RFC 9293 / STD 7)

Powyżej gniazdek: protokoły autokonfiguracji

- BOOTP (oryginalna wersja: RFC 951)
- DHCP (oryginalna wersja: RFC 1531)

Proszę zwrócić uwagę na wbudowaną w oba protokoły możliwość ich rozszerzania. Proszę też zwrócić uwagę, że to nie wystarczyło, aby dostosować BOOTP do współczesnych wymagań i wprowadzono DHCP.

Powyżej gniazdek: DNS (Domain Name System)

- RFC 1035 (i późniejsze).
- Hierarchiczne nazwy domen.
- System rozproszony, każda poddomena może mieć odrębny zbiór tzw. serwerów autorytatywnych.
- Powszechnie znane adresy zbioru serwerów odpowiedzialnych za korzeń drzewa domen.
- Z każdą nazwą w DNS związany jest zbiór rekordów; ich typy są zdefiniowane w odpowiednich RFC.
- Najważniejsze typy: SOA wraz z NS, A, PTR, MX.
- Tzw. resolver i lokalne serwery cache'ujące.

Korzystanie z DNS (dawne podejście)

Zanim powstał DNS, każdy komputer miał plik `/etc/hosts` z adresami IP i odpowiadającymi im nazwami:

```
127.0.0.1       localhost
203.0.113.1     example-uni-gw
203.0.113.28    example-uni-ftp
```

Funkcje `gethostbyname` i `gethostbyaddr` z biblioteki C wyszukiwały w nim informacje.

Te funkcje rozszerzono tak, aby odwoływały się do DNS jeśli w `/etc/hosts` nie znajdą pasującej linii.

Korzystanie z DNS (współcześnie)

Obecnie zalecane jest korzystanie z funkcji `getaddrinfo` i `getnameinfo`. Operują na pełnych strukturach `sockaddr_in` lub `sockaddr_in6`, a nie tylko na adresach IP.

Używane jest pojęcie usługi, obejmujące numer portu i protokół transportowy. Np. dla `"http"` zwrócone zostanie 80 / `IPPROTO_TCP`, a dla `"echo"` 7 / `IPPROTO_TCP` oraz 7 / `IPPROTO_UDP`.

Pełny dostęp do wszystkich rekordów w DNS możliwy jest np. za pomocą resolvera będącego częścią GNU C Library. Nie ma on zbyt obfitej dokumentacji, strona resolver(3) w manualu tylko wylicza dostępne funkcje i zakłada, że czytelnik już wcześniej dobrze znał DNS.

Powyżej gniazdek: poczta elektroniczna

- Podstawowy format listu elektronicznego (RFC 822).
- Przesyłanie listu od serwera do serwera: Simple Mail Transfer Protocol (SMTP, RFC 821).
- Alfabety narodowe, multimedia, załączniki: Multipurpose Internet Mail Extensions (MIME, RFC 1521 i 1522).
- Warta zapamiętania część MIME: nagłówek `Content-Type`.
- Zdalny dostęp do własnej skrzynki pocztowej: POP albo IMAP.

Adresy e-mail to, formalnie rzecz biorąc, adresy z warstwy 7.

Powyżej gniazdek: pajęczyna WWW

- Hypertext Markup Language 2.0 (HTML, RFC 1866).
- Hypertext Transfer Protocol 1.0 (HTTP, RFC 1945).

Następne wersje HTML i związanych z nim standardów rozwijane były już w ramach W3C; HTTP pozostał pod opieką IETF.

HTTP jest bardzo wdzięcznym przykładem pokazującym jak protokół potrafi ewoluować w odpowiedzi na nowe wymagania, które przed nim się stawia.

HTTP: podstawowe założenia

- Pojęcie zasobu.
- Lokalizatory zasobów, czyli URL-e (RFC 1738).

  scheme://user:password@host:port/url-path
- Najczęstszą operacją jest pobranie zasobu z serwera (tzw. GET), ale przewidziano również inne operacje.
- HTTP to mieszany protokół tekstowo-binarny (w zapytaniu i w odpowiedzi nagłówki są tekstowe, ciało może być binarne).

HTTP: najprostsze zapytanie

Przeglądarka chce pobrać http://www.example.org/cytaty/ala.txt
i wysyła do serwera zapytanie:

```
GET /cytaty/ala.txt HTTP/1.0
User-Agent: Mozilla/1.0
```

(po nagłówku jest pusta linia, a po niej mogłoby być ciało zapytania, gdyby to było coś innego niż GET). Serwer odpowiada:

```
HTTP/1.0 200 OK
Server: Apache/1.3.3
Content-Type: text/plain
Content-Length: 14
Last-Modified: Thu, 1 Mar 2018 12:01:01 GMT

Ala ma kota.
```

Serwery HTTP

- Apache, ISS, nginx — znane i popularne.
- Oprócz statycznych plików potrafią zwracać dokumenty dynamicznie generowane przez programy napisane w PHP, Javie, itd.
- Duże dynamiczne witryny często mają wiele bliźniaczych serwerów (tzw. backendów) schowanych za pojedynczym reverse proxy, bo tylko w ten sposób można osiągnąć niezbędną wydajność.
- Serwery obsługują równocześnie wielu klientów, zazwyczaj w wielu podprocesach czy też wątkach.

Równoległa obsługa klientów — zarys

Pseudokod znanego już nam serwera iteracyjnego (jeden klient na raz):

```
while (true) {
    clnt_sock = accept(serv_sock);
    read/write(clnt_sock);
    close(clnt_sock);
}
```

Tzw. serwer forkujący tworzy po `accept()` nowy podproces (albo wątek), przekazuje mu `clnt_sock` do obsłużenia, a sam wraca na początek pętli. Wadą może być liczba tworzonych podprocesów/wątków.

Serwer bazujący na zdarzeniach czeka aż jądro systemu da znać, że któreś z gniazdek weszło w stan gdzie można odebrać nowe połączenie lub odczytać/zapisać dane. Wykonuje odpowiednie akcje i wraca do czekania. Skomplikowane w implementacji, warto użyć abstrakcji deterministycznego automatu skończonego.

HTTP: bardziej skomplikowane zapytania

- Podstawowe metody GET, HEAD, POST, dodatkowe metody PUT, DELETE i wiele innych, patrz np. WebDAV (RFC 4918).
- Pojęcie idempotentności metody.
- Autoryzacja użytkowników (status 401 z polem WWW-Authenticate, pole Authorization w ponowionym zapytaniu).
- Kompresja zwracanej odpowiedzi (pole Content-Encoding).
- Negocjacja typu dokumentu (pola Accept, Accept-Encoding, itd.)

  Accept: text/html, text/plain; q=0.8, text/*; q=0.5, */*; q=0.1
- Śledzenie połączeń między stronami (pole Referer).
- Cache'owanie i odświeżanie zwróconych odpowiedzi (Last-Modified, If-Modified-Since, Expires).

HTTP: rozszerzenia

HTTP można rozszerzać definiując nowe metody i pola nagłówków (w tym takie, które powodują przejście na zupełnie inny protokół warstwy 7):

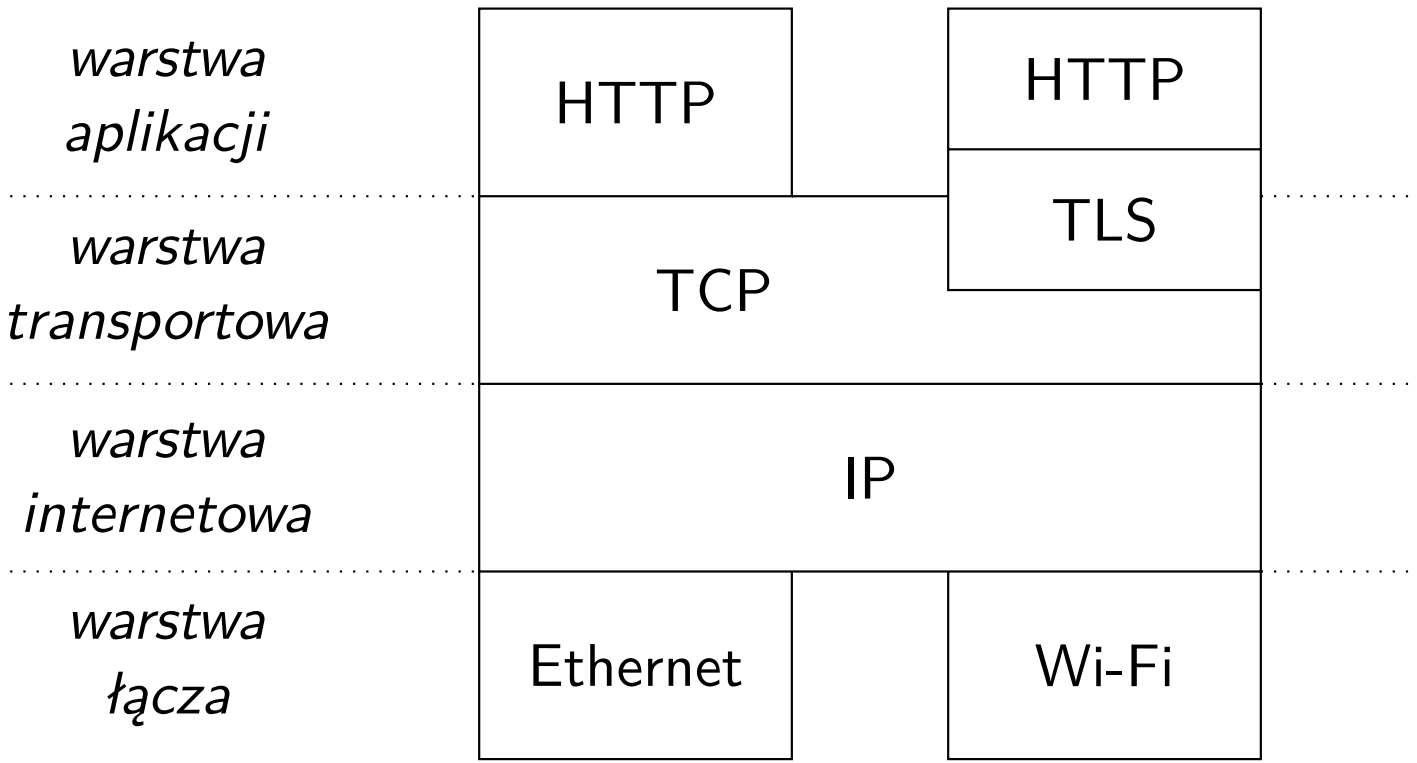
- wirtualne serwery (pole Host nagłówka)
- połączenia trwałe (Keep-Alive)
- grupowanie połączeń w sesje (Set-Cookie, Cookie)
- pobieranie fragmentów dokumentu (Range)
- serwery proxy (skomplikowany temat obejmujący wybór proxy, wskazanie docelowego zasobu, ewentualną autentykację, zarządzanie cache'owaniem (w tym Pragma: no-cache), i tak dalej)
- transparentne proxy
- reverse proxy
- protokół WebSocket

HTTPS, czyli szyfrowanie transmisji

- Nie wszyscy operatorzy telekomunikacyjni są godni zaufania, mogą podsłuchiwać i modyfikować przechodzące przez ich routery dane.
- Nieakceptowalne jeśli chcemy mieć bankowość internetową.
- Wymyślono więc SSL (Secure Sockets Layer), obecnie znany jako TLS (Transport Layer Security):
  - Strumieniowy protokół transportowy korzystający nie z protokołu sieciowego, lecz innego transportowego.
  - Przy zestawianiu połączenia („tunelu”) generowany jest klucz jednorazowy, służący potem do szyfrowania danych aplikacji.
  - Serwer przedstawia się certyfikatem, klient weryfikuje czy został on wystawiony przez jeden z oficjalnych urzędów.
  - Serwer udowadnia, że posiada tajny klucz związany z tym certyfikatem, co potwierdza jego tożsamość zapisaną w certyfikacie.
- HTTPS to HTTP w tunelu TLS w połączeniu TCP.

HTTPS, czyli szyfrowanie transmisji

TLS nie daje się w 100% wpasować w internetowy model warstwowy (w model ISO/OSI też nie):



Szerzej o tym, jak działa TLS będzie w późniejszej części wykładu.

Standardowe sposoby przesyłania struktur danych

Wracamy do przykładu, w którym klient wysyła ciągi liczb, a serwer oblicza i zwraca sumy tych ciągów.

Przed przystąpieniem do pisania kodu trzeba było zdefiniować protokół komunikacyjny, czyli ustalić czym jest liczba a czym ciąg i jak się je przekłada na przesyłane przez sieć sekwencje bajtów. Nawet w tak prostym przykładzie było to dość żmudne. Prościej byłoby odwołać się do jakiegoś standardu zawierającego definicje typów danych i ich sieciowych reprezentacji.

W powszechnym użyciu jest wiele takich standardów. Wykład omawia trzy z nich, dobrane tak, aby ilustrowały różne podejścia do problemu kodowania struktur danych na potrzeby przesyłania ich przez sieć.

Standardowe sposoby przesyłania struktur danych

Dwa stare, ale ciągle będące w użyciu, binarne standardy:

- XDR (External Data Representation Standard)
  opracowany przez firmę Sun w latach 80, RFC 1014 z 1987 r.
  RFC 4506 / STD 67 to jego najnowsza wersja
- ASN.1 z kodowaniem DER
  (Abstract Syntax Notation One, Distinguished Encoding Rules)
  też z lat 80, dzieło CCITT, obecnie standard ITU-T/ISO

Standard tekstowy:

- JSON (JavaScript Object Notation)
  początek XX wieku, początkowo nieformalny podzbiór JavaScriptu
  RFC 8259 / STD 90

Uwaga na marginesie: XML nie jest standardem przesyłania danych, lecz narzędziem służącym do definiowania takich standardów!

Serwer dodający liczby — XDR

XDR definiuje typ „liczba całkowita ze znakiem”, potrafiący wyrazić wartości z przedziału $[-2^{31}, 2^{31} - 1]$.

Wartości tego typu są kodowane jako 32-bitowe liczby całkowite w notacji uzupełnieniowej do dwóch i przesyłane jako ciąg czterech bajtów starszym bajtem najpierw (*big-endian byte order*, *network byte order*).

liczba	przesyłane bajty
2 147 483 647	127 255 255 255
1000	0 0 3 232
72	0 0 0 72
0	0 0 0 0
−1	255 255 255 255
−2	255 255 255 254
−2 147 483 648	128 0 0 0

Serwer dodający liczby — XDR

Wśród typów złożonych XDR są tablice zawierające elementy jednego, określonego typu. Tablice mogą być stałej lub zmiennej długości.

Przy kodowaniu tablicy zmiennej długości najpierw zapisuje się 4 bajty reprezentujące liczbę elementów, potem elementy tablicy (każdy z nich zajmuje tyle bajtów, ile wynika z typu tablicy).

ciąg liczb	przesyłane bajty
(7, 22, 256)	0 0 0 3 0 0 0 7 0 0 0 22 0 0 1 0
(+1, −1)	0 0 0 2 0 0 0 1 255 255 255 255
( )	0 0 0 0

Język XDR

Opis w języku naturalnym („klient przesyła serwerowi tablicę liczb całkowitych mającą nie więcej niż 100 elementów”) jest czytelny dla prostych przypadków, ale nie dla bardziej skomplikowanych („struktura zawiera tablicę identyfikatorów użytkowników i tablicę odpowiadających im struktur zawierających...”).

Nawet w prostych przypadkach łatwo o niejasności — czy te liczby w tablicy są ze znakiem, czy bez?

Formalny język opisujący typy danych zbudowane w oparciu o prymitywy XDR jest praktyczną koniecznością. Specyfikacja XDR definiuje taki język, wzorowany na języku C.

„Klient przesyła serwerowi argument typu `int<100>`” jest krótkie i jednoznaczne.

Język XDR — typy proste

- liczby całkowite 32- i 64-bitowe

  `int` *identyfikator*;
  `unsigned int` *identyfikator*;
  `hyper` *identyfikator*;
  `unsigned hyper` *identyfikator*;
- liczby zmiennoprzecinkowe 32-, 64- i 128-bitowe

  `float` *identyfikator*;
  `double` *identyfikator*;
  `quadruple` *identyfikator*;

Sposób kodowania liczb zmiennoprzecinkowych zaczerpnięty jest ze specyfikacji IEEE, wszystkie współczesne CPU obsługują ten format.

Język XDR — typy proste

- typy wyliczeniowe

  `enum { `*nazwa* = *stała*, ... `} `*identyfikator*;

  (np. `enum { NORTH = 1, EAST = 2, SOUTH = 4, WEST = 8 }`)
- wartości logiczne

  `bool` *identyfikator*;

  (skrótowy zapis dla `enum { FALSE = 0, TRUE = 1 }` *identyfikator*)

Tak samo jak w języku C typy wyliczeniowe są pochodnymi typu `int`, co za tym idzie są kodowane na 4 bajtach.

Język XDR — typy proste

- ciągi bajtów stałej lub zmiennej długości

  `opaque` *identyfikator*`[`*długość*`];`
  `opaque` *identyfikator*`<`*maksymalna_długość*`>;`
  `opaque` *identyfikator*`<>;`
- łańcuchy, czyli ciągi znaków ASCII zmiennej długości

  `string` *identyfikator*`<`*maksymalna_długość*`>;`
  `string` *identyfikator*`<>;`
- typ pusty

  `void`

Domyślna maks. dł. to $2^{31} - 1$. Zakodowana postać ciągów i łańcuchów jest w razie potrzeby uzupełniana dodatkowymi bajtami wartości 0 tak, aby łączna liczba bajtów była wielokrotnością 4.

Język XDR — konstrukcje pomocnicze

- definicje stałych

  `const` *nazwa* = *wartość*;

  ```
  const LICZBA_MICKIEWICZA = 44;
  ```

- definicje nowych typów

  `typedef` *deklaracja*;

  ```
  typedef string login_name<8>;
  typedef enum { NAY = 0, YEA = 1 } vote;
  ```

Dla typów wyliczeniowych można użyć skróconego zapisu:

```
enum vote { NAY = 0, YEA = 1 };
```

Tak samo będzie w przypadku struktur i unii.

Język XDR — typy złożone

- tablice ustalonej lub zmiennej długości

  typ_elementów identyfikator`[`*długość*`]`;
  typ_elementów identyfikator`<`*maksymalna_długość*`>`;
  typ_elementów identyfikator`<>`;

- rekordy (struktury)

```
struct {
    deklaracja_pola;
    ...
} identyfikator;
```

```
struct person {
    string given_name<63>;
    string family_name<255>;
    int birth_year;
};
```

Język XDR — typy złożone

- rekordy z wariantami (tagged/discriminated union)

  `union switch (` *dekl_pola_znacznikowego* `) {`
  `case` *wartość_A*:
  deklaracje_pól_A;
  …
  `}` *identyfikator*;

  oprócz gałęzi `case` *wartość* może wystąpić gałąź `default`

```
union server_reply switch (bool success) {
    case TRUE:
        opaque result[32];
    case FALSE:
        unsigned int error_code;
        string error_message<255>;
};
```

Język XDR — typy złożone

- dane opcjonalne (mogą, ale nie muszą wystąpić)
  typ * *identyfikator*;

Równoważne deklaracji
  typ identyfikator<1>;
czyli tablicy mogącej mieć 0 lub 1 element.

Przydatne przy definiowaniu typów mających odpowiadać listom pojedynczo wiązanym, drzewom binarnym oraz innym abstrakcyjnym typom danych, w których nie może być kilku wskaźników pokazujących na to samo miejsce w pamięci.

Język XDR — przykłady zastosowań

NFS, czyli sieciowy system plików opracowany przez firmę Sun a później rozwijany przez IETF używa XDR do definiowania struktur danych przekazywanych pomiędzy klientem a serwerem (patrz RFC 7531).

sFlow to standard monitorowania infrastruktury sieciowej. Switche i routery przesyłają do stacji zarządzającej losowo wybrane próbki przechodzących przez nie pakietów oraz statystyki ruchu. Format tych danych: patrz `https://sflow.org/developers/specifications.php`.

Pomocnicze biblioteki kodujące dane

Proces kodowania wartości zapisanych w zmiennych programu do postaci ciągu bajtów znany jest jako serializacja; proces odwrotny to deserializacja.

Można impl. je własnoręcznie, ale to rutynowy, banalny kod. Na 32- lub 64-bitowym systemie operacyjnym wartość XDR-owego typu `int` można zapisać w zmiennej języka C typu `int`, typowi `string` będzie odpowiadać zmienna `char *` albo `char []`, itd. W innych językach będzie podobnie.

Wielokrotnie wykorzystywane fragmenty kodu powinny być w bibliotece. Dla języka C bibliotekę XDR opracowała i udostępniła firma Sun, jest częścią glibc; Python miał standardowy, dość ubogi moduł xdrlib; Java w domyślnej dystrybucji nie ma niczego. Biblioteki zazwyczaj zawężają nieco naszą swobodę (np. deserializacja łańcuchów tylko do zmiennych `char *` i zawsze przy tej okazji alokowany jest nowy obszar pamięci).

Biblioteka Sun XDR

Biblioteka oparta jest o dwie abstrakcje:

- strumień XDR zawierający ciąg bajtów
- funkcje filtrujące, które czytają/zapisują do strumienia wartości określonego typu
- ... oraz potrafią zdealokować pamięć przydzieloną dla wczytanych łańcuchów, tablic zmiennej długości, itp.

Przykładowa funkcja filtrująca:

```
bool_t xdr_int(XDR * xdrs, int * i);
```

W zależności od tego czy strumień przekazany jako pierwszy argument jest w trybie „do odczytu” czy „do zapisu”, funkcja modyfikuje bądź pobiera wartość pamiętaną w zmiennej języka C przekazanej jako drugi argument. Zwracane jest TRUE w razie sukcesu albo FALSE w razie wystąpienia błędu (strumień jest wtedy w nieokreślonym stanie, nie należy na nim wykonywać dalszych operacji we-wy).

Biblioteka Sun XDR — serializacja

```
FILE * fp = fopen("dane_xdr.bin", "wb");

XDR xs;
xdrstdio_create(&xs, fp, XDR_ENCODE);

bool_t wszystko_ok = TRUE;
int i = 13;
wszystko_ok = wszystko_ok && xdr_int(&xs, &i);
float f = 3.1415;
wszystko_ok = wszystko_ok && xdr_float(&xs, &f);
if (! wszystko_ok) {
    fprintf(stderr, "xdr: data serialization failure\n");
    exit(1);
}

xdr_destroy(&xs);   // i fclose(fp) jeśli to już wszystko
```

Biblioteka Sun XDR — deserializacja

```
#include <rpc/xdr.h>

char buf[1024];
ssize_t len = recvfrom(sock, buf, sizeof(buf),
                                        0, NULL, NULL);

XDR xs;
xdrmem_create(&xs, buf, len, XDR_DECODE);

char * s = NULL;
double d;
bool_t ok;
ok = xdr_string(&xs, &s, 255) && xdr_double(&xs, &d);

⋮
```

Biblioteka Sun XDR — deserializacja

```
⋮

if (! ok) {
    fprintf(stderr, "xdr: deserialization failure\n");
    exit(1);
}
unsigned int pos = xdr_getpos(&xs);
if (pos != len) {
    fprintf(stderr, "xdr: extra bytes in datagram\n");
}

xdr_destroy(&xs);

printf("%s %f\n", s, d);
xdr_free(xdr_string, &s);   // czyli free(s); s = NULL;
xdr_free(xdr_double, &d);   // a to jest operacją pustą
```

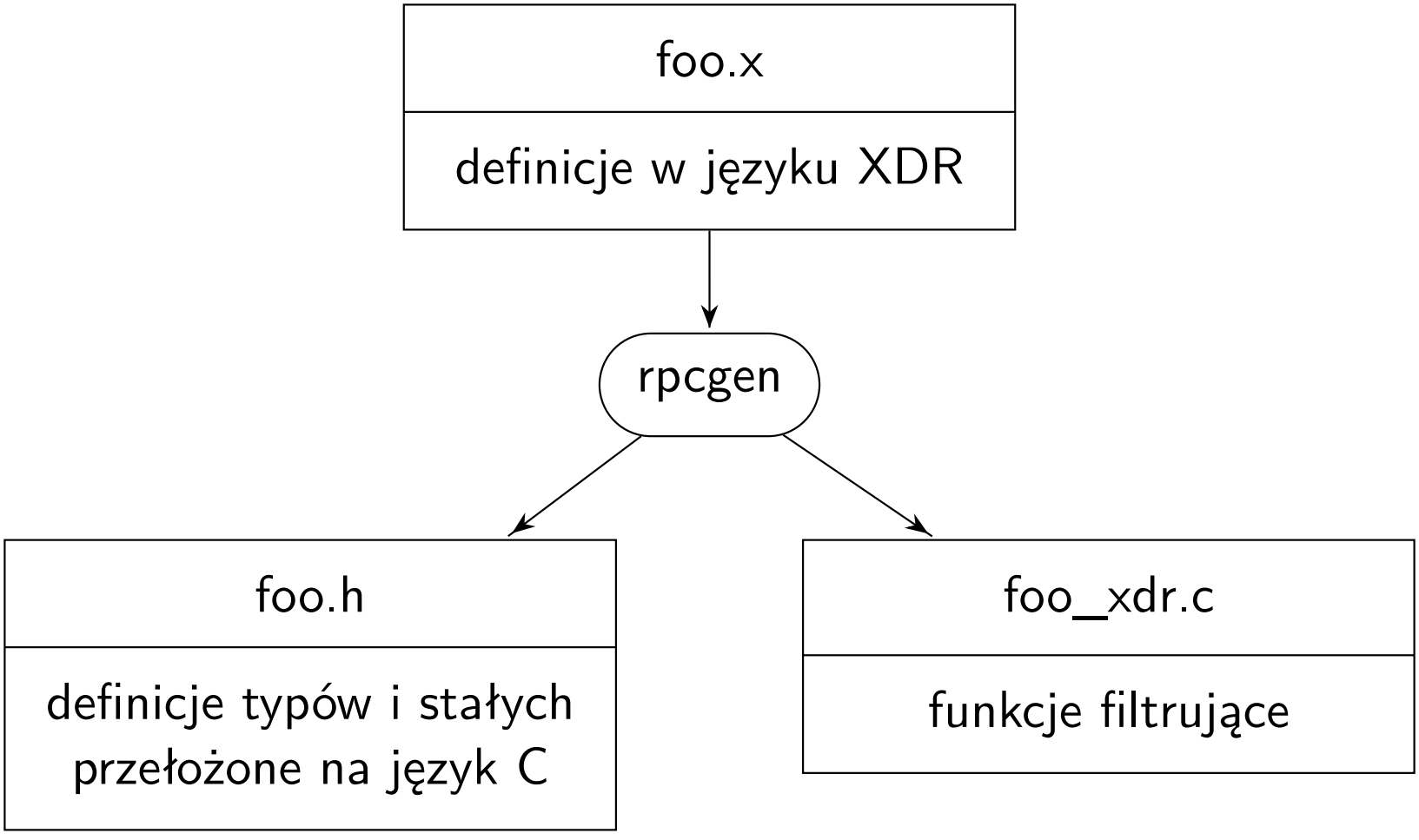
Biblioteka Sun XDR — filtry użytkownika

Biblioteka dostarcza filtry dla wszystkich standardowych typów; w oparciu o nie implementowane są filtry dla typów zdef. przez użytkownika.

```
// deklaracje w języku XDR

typedef string login_name<8>;
struct punkt { float x, y; };

// funkcje-filtry w języku C

bool_t xdr_login_name(XDR * xp, char * * p) {
    return xdr_string(xp, p, 8);
}
bool_t xdr_punkt(XDR * xp, struct punkt * p) {
  return xdr_float(xp, &(p->x)) && xdr_float(xp, &(p->y));
}
```

Biblioteka Sun XDR — filtry użytkownika

Filtrów nie implementuje się ręcznie, generuje je pomocniczy program rpcgen w oparciu o plik *.x z definicjami stałych i typów.



Alternatywne biblioteki

Przedstawiona powyżej biblioteka procedur XDR jest częścią większej biblioteki znanej jako Sun RPC lub ONC RPC (stąd nazwa „rpcgen").

TI-RPC to jej rozszerzona wersja, potrafiąca korzystać z IPv6.

ONCRPC4J oraz JRPC to biblioteki Javy, mają oczywiście swoje własne narzędzia oncrpc4j-rpcgen.jar i jrpcgen.exe generujące pliki *.java na podstawie plików *.x.

Biblioteka RemoteTea.Net jest dla języka C#.

Dla języka Go jest kilka pakietów (sunrpc, oncrpc, xdr2, ...) o różnym stopniu kompletności.

Wszystkie akceptują ten sam język specyfikacji typów danych. Można napisać serwer w jednym języku, kilka programów klienckich w innych językach i mieć pewność, że ich procedury serializacji/deserializacji wymienianych danych będą kompatybilne, bo zostały wygenerowane z tego samego pliku *.x.

Serwer dodający liczby — DER

Wróćmy do dodawania ciągów liczb i spróbujmy zdef. przesyłane ciągi bajtów odwołując się do standardu DER.

DER koduje dane używając schematu TLV (type-length-value). Zapisuje kolejno: znacznik z numerycznym identyfikatorem typu, liczbę bajtów zajętych przez zakodowaną wartość, wartość zakodowaną w postaci ciągu bajtów zgodnie z regułami obowiązującymi dla jej typu.

Dane zakodowane z użyciem DER są więc, przynajmniej częściowo, samoopisujące się. Można przeanalizować przechwycony datagram i stwierdzić, że np. zawiera on dwie liczby rzeczywiste, wartość jakiegoś typu mającego niestd. znacznik 161 i tablicę z pięcioma łańcuchami. Dla kodowania XDR taka analiza nie byłaby możliwa.

Serwer dodający liczby — DER

ASN.1 oferuje typ INTEGER (o nieograniczonym zakresie).
Znacznik typu INTEGER to 2. DER zapisuje liczby całkowite w notacji uzupełnieniowej do dwóch starszym bajtem najpierw.

liczba	przesyłane bajty
5 000 000 000	2 5 1 42 5 242 0
1000	2 2 3 232
128	2 2 0 128
127	2 1 127
72	2 1 72
0	2 1 0
−1	2 1 255
−2	2 1 254
−128	2 1 128
−129	2 2 255 127

Serwer dodający liczby — DER

Podstawowym typem złożonym w ASN.1 jest SEQUENCE, reprezentująca ciąg wartości. Można go użyć do reprezentowania i struktur, i tablic. Znacznik typu SEQUENCE to 48.

ciąg liczb	przesyłane bajty
(7, 22, 256)	48 10 2 1 7 2 1 22 2 2 1 0
(+1, −1)	48 6 2 1 1 2 1 255
( )	48 0

Serwer dodający liczby — DER

Certyfikaty TLS to pliki w formacie DER (częściej DER z dodatkowym kodowaniem PEM), biblioteka OpenSSL ma więc narzędzie diagnostyczne potrafiące analizować pliki DER. Użyjmy go do sprawdzenia, czy przykład z poprzedniego slajdu jest poprawny.

```
$ od -A d -t u1 dane_der.bin
0000000  48  10   2   1   7   2   1  22   2   2   1   0
0000012
$ openssl asn1parse -i -inform DER -in dane_der.bin
    0:d=0  hl=2 l=  10 cons: SEQUENCE
    2:d=1  hl=2 l=   1 prim:  INTEGER           :07
    5:d=1  hl=2 l=   1 prim:  INTEGER           :16
    8:d=1  hl=2 l=   2 prim:  INTEGER           :0100
```

„d” to poziom zagnieżdżenia, „hl” długość nagłówka (znacznik+długość), „l” długość wartości właściwej

Język ASN.1

Standard ASN.1 dostarcza języka pozwalającego definiować typy danych oraz specyfikować instancje tych typów (tzn. wartości). ASN.1 nie określa w jaki sposób wartości mają być serializowane — za to odpowiadają pomocnicze standardy, DER jest jednym z nich.

```
Liczba ::= INTEGER
CiagLiczb ::= SEQUENCE SIZE (1..25) OF INTEGER

Punkt2D ::= SEQUENCE { x REAL, y REAL }

ProcentAlkoholu ::= INTEGER (0..100)
Napitek ::= SEQUENCE {
    nazwa UTF8String,
    moc ProcentAlkoholu DEFAULT 0
}
```

Język ASN.1

Kilka instancji typów z poprzedniego slajdu:

```
dwa Liczba ::= 2
liczby CiagLiczb ::= { 7, 22, 256 }

a Punkt2D ::= { x 1.5, y 7 }
b Punkt2D ::= { x -20, y 0.123 }

wodka Napitek ::= { nazwa "Wyborowa", moc 45 }
mleko Napitek ::= { nazwa "Łaciate" }
```

Język ASN.1

Język ASN.1, tak samo jak język XDR, gwarantuje jednoznaczność spisanych w nim specyfikacji i z tego powodu często jest wykorzystywany w standardach telekomunikacyjnych i sieciowych.

Dostępne są narzędzia, które na podstawie specyfikacji generują kod (de)serializujący zdef. w niej typy. Jest to bardziej skomplikowany proces niż w przypadku XDR, bo ASN.1 ma większą siłę wyrazu (ograniczenia na wartości pól, pola opcjonalne, pola z domyślnymi wartościami itd.).

Ręczne pisanie kodu, który np. wczytywałby certyfikaty SSL sprawdzając czy są zgodne z def. podaną w RFC 2459 (str. 15 i dalsze) jest zdecydowanie niewskazane — patrz następny slajd aby się przekonać o złożoności tego problemu.

Wewnętrzna struktura certyfikatu SSL/TLS

```
$ openssl asn1parse -inform PEM -in witryna.crt
 0:d=0  hl=4 l= 891 cons: SEQUENCE
 4:d=1  hl=4 l= 611 cons:  SEQUENCE
 8:d=2  hl=2 l=   3 cons:   cont [ 0 ]
10:d=3  hl=2 l=   1 prim:    INTEGER :02
13:d=2  hl=2 l=   9 prim:   INTEGER :B3C8C4D58A88FFDA
24:d=2  hl=2 l=  13 cons:   SEQUENCE
26:d=3  hl=2 l=   9 prim:    OBJECT :sha256WithRSAEncryption
37:d=3  hl=2 l=   0 prim:    NULL
39:d=2  hl=2 l=  84 cons:   SEQUENCE
41:d=3  hl=2 l=  11 cons:    SET
43:d=4  hl=2 l=   9 cons:     SEQUENCE
45:d=5  hl=2 l=   3 prim:      OBJECT :countryName
50:d=5  hl=2 l=   2 prim:      PRINTABLESTRING :PL
54:d=3  hl=2 l=  19 cons:    SET
.....
```

Serwer dodający liczby — JSON

JSON to format tekstowy. Przesyłane ciągi bajtów są interpretowane jako znaki w kodowaniu UTF-8, a następnie przekazywane do parsera robiącego analizę składniową, dokładnie tak samo jak przy kompilowaniu kodu źródłowego w jakimś języku programowania.

Parser ignoruje białe znaki, daną strukturę danych można więc wyrazić jako ciąg bajtów na nieskończenie wiele sposobów.

Wartości proste rozpoznawane przez parser: liczby, łańcuchy, wartości logiczne (`true` oraz `false`), wartość pusta `null`.

Wartości złożone: tablice (elementy mogą być różnych typów) oraz tzw. obiekty, czyli tak naprawdę słowniki (znane również jako tablice asocjacyjne).

Serwer dodający liczby — JSON

Standard JSON nie definiuje formalnego języka, w którym można byłoby specyfikować typy zapytań i odpowiedzi przetwarzanych przez serwer. Często używa się więc języka naturalnego: zapytanie to tablica liczb; odpowiedź to albo liczba, albo łańcuch `"ERROR"`; wszystkie przesyłane liczby muszą być całkowite oraz nieujemne.

Biblioteki JSON przy deserializacji ciągu bajtów sprawdzają jego poprawność składniową, ale nie semantyczną. Weryfikacja tego, czy klient przysłał tablicę, czy wszystkie jej elementy są liczbami itd. musi więc zostać ręcznie zaimplementowana.

Serwer dodający liczby — JSON Schema

Strukturę przesyłanych danych można, zamiast w języku naturalnym, spróbować wyrazić jako tzw. schemat. Schematy zapisuje się w plikach w formacie JSON.

Schemat dla tablicy nieujemnych liczb całkowitych:

```
{
    "type": "array",
    "items": {
        "type": "integer",
        "minimum": 0
    }
}
```

Użycie JSON Schema do walidacji odbieranych z sieci danych wymaga odwołania się do dodatkowych bibliotek.

Serwer dodający liczby — JSON Schema

Decydując się na użycie JSON Schema trzeba pamiętać o niestabilności tego standardu. Jego wstępna wersja została opublikowana w 2013, przez następną dekadę publikowano kolejne, niekompatybilne ze sobą wersje. Plan dalszych prac ogłoszony w 2023 mówi o jeszcze jednym złamaniu kompatybilności, które podobno ma już być ostatnie.

Automatyczne generowanie rutynowego kodu sieciowego

Skoro istnieją narzędzia generujące kod (de)serializujący dane, to może da się też zautomatyzować tworzenie innych fragmentów kodu, np. tych odpowiadających za obsługę gniazdek sieciowych?

Dwa następne slajdy zawierają listy typowych rzeczy robionych przez serwer i przez klientów. Czynności rutynowe są na czarno — jak widać, jest ich zdecydowana większość.

Czynności wykonywane przez serwer

1. Stwórz nasłuchujące gniazdko sieciowe.
2. Czekaj aż klient nawiąże połączenie.
3. Odbierz przesłany ciąg bajtów i go zdeserializuj.
4. Oblicz wynik na podstawie zdekodowanych argumentów.
5. Zakoduj wynik do postaci ciągu bajtów i odeślij go klientowi.
6. Wróć do oczekiwania na nowe połączenie.

Rutynowe czynności oczywiście obejmują obsługę błędów.

Mogą też być bardziej skomplikowane, niż sugeruje to powyższa lista. Serwer może np. otwierać dwa gniazdka, TCP oraz UDP, co pozwala klientom wybierać połączeniowy bądź datagramowy sposób komunikacji.

Czynności wykonywane przez klientów

1. Ustal adres serwera (np. na podstawie argv).
2. Wczytaj liczby, które chcemy dodać (np. z pliku).
3. Nawiąż połączenie z serwerem.
4. Zserializuj strukturę danych zawierającą liczby.
5. Wyślij zakodowane dane, poczekaj na odpowiedź.
6. Zdekoduj odpowiedź.
7. Zrób coś ze zwróconym wynikiem (np. go wyświetl).

Serwer zazwyczaj potrafi wykonać kilka różnych rzeczy (np. oprócz sum może obliczać iloczyny). Trzeba wtedy wprowadzić pojęcie identyfikatora wywoływanej operacji i wysyłać go przed zakodowanymi argumentami.

Zdalne wywołanie procedury

Wiele usług sieciowych działa na takiej samej zasadzie, jak nasz sumator liczb — podaje im się argumenty, a one obliczają i zwracają wynik. Bardzo to przypomina zwykłe wywołanie procedury/funkcji, stąd nazwa *remote procedure call* (RPC).

Niektórzy teoretycy zajmujący się inżynierią oprogramowania proponowali, aby zdalne wywołania procedur traktować tak samo jak lokalne. Miało to uprościć tworzenie aplikacji rozproszonych. W praktyce różnic między nimi jednak nie daje się zignorować; jedną z ważniejszych jest ryzyko, że zdalna procedura nie zwróci wyniku jeśli akurat sieć padła.

Standardy RPC

Na rynku jest wiele standardów RPC. Każdy z nich:

- definiuje sposób serializowania argumentów i wyników procedur
- definiuje pojęcie adresu/nazwy procedury (konieczne jeśli słuchający na danym gniazdku serwer obsługuje więcej niż jedną)
- definiuje protokół RPC, czyli postać komunikatów reprezentujących wywołanie procedury i zwrócenie wyniku albo błędu
- określa jak te komunikaty są przesyłane przy pomocy protokołu/-ów transportowych niższego rzędu
- (zazwyczaj) definiuje język, w którym można formalnie specyfikować jakie procedury są dostępne, jakie są typy ich argumentów i wyników, itd. — jest to tzw. interface description language (IDL)

Dla danego standardu może istnieć wiele bibliotek go implementujących. Mogą się różnić sposobem, w jaki przekładają definicje IDL na język, w którym pisana jest aplikacja, mogą w różny sposób generować rutynowe fragmenty kodu, ale potrafią się ze sobą porozumieć.

Standard Sun RPC (znany też jako ONC RPC)

Opracowany w latach 80 jako podstawa dla rozwiązań, które stały się uniksowym standardem: NFS (sieciowy system plików), YP (centralna baza kont i innych danych tradycyjnie trzymanych w /etc/), itd.

Oryginalnie używał UDP/IPv4 oraz TCP/IPv4; w późniejszych wersjach standardu wprowadzono obsługę dodatkowych protokołów transportowych. Aby móc z nich korzystać trzeba przesiąść się z oryginalnej biblioteki Sun RPC na bibliotekę TI-RPC.

Jako IDL wykorzystywana jest rozszerzona wersja XDRL, znana pod nazwą RPCL. Pozwala ona definiować tzw. programy, składające się z określonego zbioru procedur. Program może występować w kilku niezależnych od siebie wersjach — pozwala to rozwijać usługi sieciowe z zachowaniem wstecznej kompatybilności.

Przykładowa specyfikacja w języku RPC

```
typedef unsigned int sum_argument<100>;

union sum_wynik switch (bool wynik_jest_ok) {
    case TRUE:
        unsigned int suma;
    case FALSE:
        void;
};

program SUM_PROGRAM {
    version SUM_WERSJA_1 {
        sum_wynik SUM_OBLICZ(sum_argument) = 1;
    } = 1;
} = 0x20002019;
```

Adresowanie zdalnych procedur

Każdy program, wersja i procedura mają 32-bitowy identyfikator. Oprócz nich w adresie zdalnej procedury występuje również nazwa bądź adres IP komputera, na którym działa proces implementujący tę procedurę.

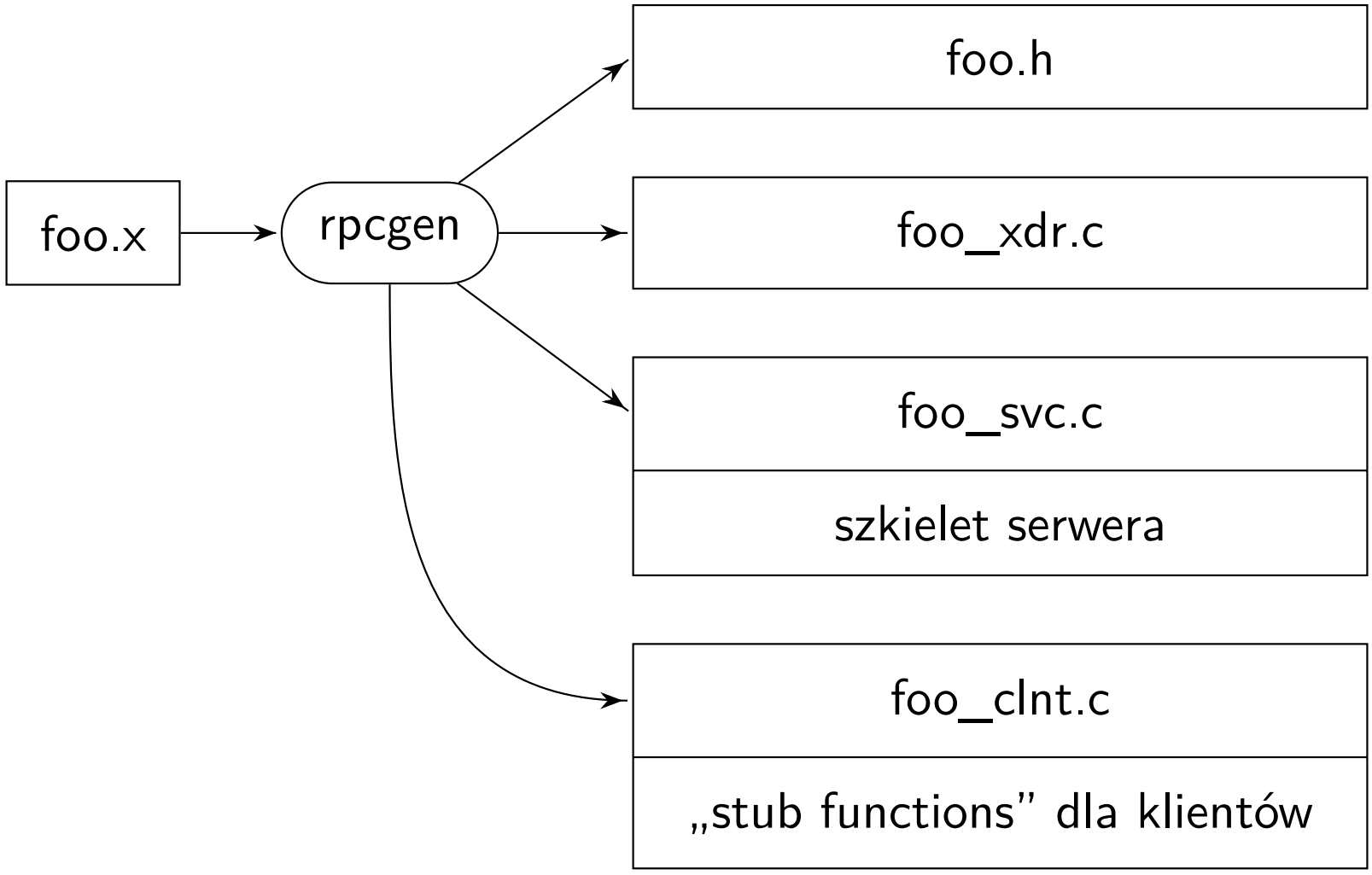
Przykład: „wywołaj w programie nr 100003, wersja 2 działającym na backups.example.org procedurę nr 9 używając UDP jako transportu".

Trzeba wysłać pakiet UDP, ale na jaki numer portu? Sun RPC zakłada obecność pomocniczej usługi, tzw. portmappera, który na pytanie „na którym porcie UDP znajdę program 100003v2" odpowie „na 718" albo „o ile wiem, na tym hoście nie ma takiego". Portmapper to program nr 100000, zawsze jest dostępny na porcie 111 (UDP i TCP).

Detale techniczne specyfikacji Sun RPC patrz RFC 1057 z 1988 r.

Generacja rutynowych fragmentów kodu

Jeśli plik ∗.x zawiera def. programu, to rpcgen stworzy dodatkowe pliki.



Kod serwera

Automatycznie generowany szkielet zawiera funkcję `main` tworzącą nasłuchujące gniazdka sieciowe UDP i TCP, rejestrującą w portmapperze ich numery portów, odbierającą zapytania i na podstawie trójki (prognum, versnum, procnum) wywołującą odpowiedni filtr XDR deserializujący argument wywołania, itd. W 95% przypadków te standardowe czynności są wszystkim, czego potrzebuje pisany właśnie serwer.

Programista musi tylko napisać w języku C funkcje implementujące zdalne procedury. W pliku *.h można znaleźć ich sygnatury — dla naszego przykładu byłaby to funkcja

```
sum_wynik * sum_oblicz_1_svc(sum_argument * argp,
                             struct svc_req * reqp)
```

Kod serwera

Nazwa funkcji zawiera numer wersji programu — to konieczne, bo RPCL dopuszcza te same nazwy procedur w różnych wersjach programu.

Zgodnie z konwencją poznaną już wcześniej przy okazji omawiania XDR argumenty i wyniki przekazywane są jako wskaźniki na zmienne odpowiedniego typu.

W strukturze `svc_req` można znaleźć informacje o adresie klienta i innych podobnych rzeczach. Można to wykorzystać np. do autoryzacji.

Kod klienta

Funkcje z *_clnt.c wymagają podania wartości typu `CLIENT *`, będącej uchwytem reprezentującym zdalny program. Tworzy się go tak:

```
CLIENT * cl = clnt_create("1.2.3.4",
                          SUM_PROGRAM, SUM_WERSJA_1, "udp");
if (cl == NULL) {
    clnt_pcreateerror("clnt_create");
    exit(EXIT_FAILURE);
}
```

Funkcja `clnt_create` łączy się z portmapperem na wskazanym hoście. Zwrócony przez portmapper numer portu UDP, na którym słucha proces obsługujący podany program RPC, jest zapisywany w strukturze `CLIENT`.

Kod klienta

Dla każdej ze zdalnych procedur zdefiniowanych w pliku *.x istnieje odpowiadająca jej funkcja-stub. Dla naszego przykładu jest to

```
sum_wynik * sum_oblicz_1(sum_argument * argp, CLIENT * c)
```

Funkcja-stub serializuje podany jej argument używając filtru XDR odpowiadającego typowi argumentu, wysyła do serwera zapytanie, czeka na odpowiedź i deserializuje wynik zwrócony przez zdalną procedurę. W razie wystąpienia błędu zwraca NULL.

Po zakończeniu przetwarzania zwróconego wyniku należy zwolnić przy pomocy funkcji `clnt_freeres` przydzieloną mu podczas deserializacji pamięć.

Kod klienta

Przykład wywołania zdalnej procedury:

```
unsigned int dane[3] = { 19, 102, 7 };

sum_argument arg = {
    .sum_argument_val = dane,
    .sum_argument_len = 3
};

sum_wynik * resp = sum_oblicz_1(&arg, cl);
if (resp == NULL) {
    clnt_perror(cl, "sum_oblicz_1");
    exit(EXIT_FAILURE);
}

⋮
```

Kod klienta

```
⋮

if (resp->wynik_jest_ok) {
    // zrób coś z resp->sum_wynik_u.suma
}

if (! clnt_freeres(cl,
            (xdrproc_t) xdr_sum_wynik, (char *) resp))
{
    fprintf(stderr, "clnt_freeres\n");
    exit(EXIT_FAILURE);
}

clnt_destroy(cl);
```

Dodatkowe informacje o Sun RPC

Jest możliwość dołączania danych autentykujących i autoryzujących do zapytań i odpowiedzi (ważne np. w NFS).

Zapytania mogą być wysyłane jako broadcasty UDP. Pozwala to wysłać zapytanie do wszystkich portmapperów w sieci lokalnej (tylko one mają ustalony numer portu UDP). Można to wykorzystać np. do odnajdywania usług, które na pewno działają w sieci, ale nie wiemy na którym hoście.

HTTP jako transport dla zdalnych wywołań

- Współcześnie wiele aplikacji jest javascriptowych, wykonują się wewnątrz przeglądarki WWW w tzw. „piaskownicy”
- ... gdzie do komunikacji sieciowej używać można tylko klasy XMLHttpRequest pozwalającej wysyłać zapytania do tej witryny, z której pobrano bieżącą stronę,
- ... lub klasy WebSocket (nowy protokół transp. z 2011 r., upgrade połączenia HTTP, datagramy wewnątrz poł. TCP lub TLS).
- Nawet jeśli aplikacja jest tradycyjna, to w hotelach, ogródkach kawiarnianych itd. można trafić na firewalle przepuszczające tylko połączenia wychodzące na port 80 lub 443 (HTTP, HTTPS)
- ... stąd powszechne użycie HTTP jako protokołu transportowego przy wywoływaniu zdalnych procedur bądź obiektów.
- Wiele podejść/standardów: REST, XML-RPC, SOAP, itd.

REST (Representational State Transfer)

- REST nie jest konkretnym protokołem. Jest architekturą, czyli ogólnym pomysłem na to jak projektować usługi sieciowe.
- Kluczowa obserwacja: HTTP jest protokołem wywołującym metody (GET, PUT, itd.) na zasobach adresowanych URL-ami.
- URL-e nie muszą odpowiadać plikom na dysku serwera, mogą reprezentować rekordy w bazie danych, posty na forum, itd.
- HTTP może przesyłać dane dowolnego typu, nie tylko text/html.
- Powyższe w zupełności wystarcza, aby zaimplementować wiele usług sieciowych (zwłaszcza te przypominające bazy danych)
- ... choć do innych (np. sumatora) raczej nie pasuje.
- Nie trzeba używać specjalnych bibliotek ani pomocniczych demonów.
- Trzeba zaprojektować schemat URL-i dla rzeczy przetwarzanych przez usługę oraz określić format, w jakim będzie przesyłany ich stan.

Projekt prostej usługi REST

Założenia: baza danych SQL, jedna z tabel to „osoby” z kolumnami przechowującymi imię, nazwisko, datę urodzenia itd., oraz kolumną z numerycznym identyfikatorem służącym jako klucz główny.

- GET /db/osoby/1234
  zwraca wiersz o wskazanym id w formacie text/tab-separated-values
- PUT /db/osoby/1234
  nadpisuje wskazany wiersz, do zapytania należy dołączyć dane TSV, można pominąć niektóre kolumny (wtedy ich zawartość się nie zmieni)
- DELETE /db/osoby/1234
  skasowanie wskazanego wiersza
- POST /db/osoby
  dodanie nowego wiersza, należy dołączyć dane TSV bez kolumny id, zwraca utworzony wiersz z przydzielonym mu id też jako TSV

Projekt prostej usługi REST

Inne operacje są mniej oczywiste i można je rozmaicie definiować:

- GET /db/osoby
  pobranie całej zawartości tabeli w formacie TSV
- ... albo pobranie tylko identyfikatorów wierszy jako text/plain
- ... albo tylko id, ale w formacie text/html z klikalnymi odnośnikami (tu wygodniejsze byłoby GET /db/osoby/)
- ... z możliwością stronicowania przy pomocy parametrów „limit” i „start” dołączanych do URL-a
- GET /db/osoby/search?nazwisko=Nowak
  zwraca wiersze zawierające wskazaną wartość
- GET /db/sql_query
  do zapytania muszą być dołączone dane formatu application/sql lub text/plain zawierające pojedyncze SELECT, wyniki zwracane jako TSV; w razie błędów składniowych SQL zwracany jest status 422

Rzeczywiste usługi REST

- Wiele typów rzeczy (dla forum mogą to być podfora, użytkownicy, wątki, posty, itd.).
- Hierarchiczne zależności między rzeczami mogą być odwzorowane w strukturze URL-i (GET /forum/4/thread/7890).
- Dane w formacie JSON lub XML.
- Autentykacja: często w oparciu o sesje HTTP, bo to wygodne przy implementowaniu AJAX-owych witryn (wykonujące się wstawki JavaScript są automatycznie zautentykowane).
- Autoryzacja: tylko autor posta może go usunąć, itp.
- *Rate limiting* w przypadku publicznie dostępnych usług
- ... oraz konieczność rejestracji w celu otrzymania dołączanego do zapytań tokenu identyfikującego aplikację i jej dewelopera (webmaster wie z kim się kontaktować w razie problemów).

Przykład publicznego API: `https://www.discogs.com/developers`

Implementacja

Klient REST:

- wszechstronna kliencka biblioteka HTTP (niestandardowe metody, pełny dostęp do nagłówków zapytań i odpowiedzi)
- biblioteki przetwarzające przesyłane formaty danych (o ile są potrzebne, dla text/plain mogą nie być)

Serwer REST:

- może samodzielnie przyjmować połączenia HTTP / HTTPS
- ... ale przeważnie robi to „poważny” serwer (Apacz bądź nginx), który zapytania o wybrane URL-e przekazuje do obsłużenia kodowi implementującemu daną usługę
- CGI to oryginalny standard takiego przekazywania, wpłynął na większość następnych

CGI (Common Gateway Interface)

- Nieformalnie od 1993, RFC 3875 z 2004 r.
- Kod usługi jest w odrębnym programie, zwanym skryptem CGI.
- Serwer HTTP ma w konfiguracji zdef. prefiks związany z danym skryptem; URL-e z takim początkowym fragmentem ścieżki są obsługiwane przez skrypt.
- Uruchomiony skrypt otrzymuje informacje o nagłówkach zapytania HTTP w zmiennych środowiska, a ciało zapytania na stdin.
- Skrypt na stdout wypisuje nagłówki (często tylko Content-Type), pustą linię, treść zwracanego dokumentu.
- Serwer uzupełnia nagłówki tak, aby otrzymać prawidłową odpowiedź HTTP i odsyła ją klientowi.

Najprostszy skrypt CGI

```
#include <stdio.h>

int main(void)
{
    printf("Content-Type: text/plain\r\n");
    printf("\r\n");
    printf("Hello, world!\n");
    return 0;
}
```

Zmienne CGI

Wybrane zmienne CGI:

- `REQUEST_METHOD`
- `SCRIPT_NAME` — prefiks
- `PATH_INFO` — część ścieżki URL po prefiksie
- `QUERY_STRING` — to, co w URL-u po pytajniku
- `CONTENT_TYPE`
- `CONTENT_LENGTH`
- `HTTP_*` — po jednej zmiennej dla każdego nagłówka HTTP w zapytaniu (`HTTP_COOKIE`, `HTTP_USER_AGENT`, itd.)

Powyższe zmienne pojawiają się też w innych standardach i językach, np. w pythonowym WSGI lub w języku PHP.

XML-RPC

- REST koncepcyjnie przypomina programowanie obiektowe, tu zaś model działania przypomina wywoływanie funkcji/procedur.
- Zdalną procedurę wywołuje się wysyłając zapytanie POST z dołączonym dokumentem XML zawierającym nazwę procedury i jej argumenty.
- Serwer też zwraca dokument XML (wynik lub komunikat o błędzie).
- Oczywiste podejście, prawdopodobnie wielu programistów z niego korzystało, każdy wymyślał swój własny XML-owy format.
- Dave Winer zadał sobie trud spisania formalnej specyfikacji i jego format w 1999 r. stał się nieoficjalnym standardem.
- Specyfikacja patrz `http://xmlrpc.com/spec.md`
- Bardzo rozwlekłe zapytania i odpowiedzi.

XML-RPC: przykładowe wywołanie procedury

```
<?xml version="1.0"?>
<methodCall>
 <methodName>przelew</methodName>
 <params>
  <param>
   <value>
    <struct>
       <member>
        <name>imie</name>
        <value><string>Adam</string></value>
       </member>

⋮
```

XML-RPC: przykładowe wywołanie procedury

```
⋮
        <member>
         <name>nazwisko</name>
         <value><string>Nowak</string></value>
        </member>
      </struct>
    </value>
   </param>
   <param>
    <value><int>-2500</int></value>
   </param>
  </params>
</methodCall>
```

XML-RPC: przykładowe wyniki

```
<?xml version="1.0"?>
<methodResponse>
 <params>
  <param>
   <value><int>7418</int></value>
  </param>
 </params>
</methodResponse>
```

XML-RPC: przykładowy błąd wywołania

```
<?xml version="1.0"?>
<methodResponse>
 <fault>
  <value>
   <struct>
    <member>
     <name>faultCode</name>
     <value><int>17</int></value>
    </member>
    <member>
     <name>faultString</name>
     <value><string>Brak środków.</string></value>
    </member>
   </struct>
  </value>
 </fault>
</methodResponse>
```

XML-RPC: typy danych

Specyfikacja określa następujące typy:

- `boolean` — 0 lub 1
- `int` czy też `i4` — 32-bitowa liczba ze znakiem
- `double` — jak w C lub Javie
- `string` — początkowo łańcuch znaków ASCII, później specyfikacja zaczęła dopuszczać znaki unikodowe
- `base64` — dane binarne w kodowaniu base64 (tak jak w MIME)
- `dateTime.iso8601` — dzień i godzina, ale bez informacji o strefie czasowej (dziwne, Internet jest przecież globalny)
- `array` — tablica
- `struct` — struktura/słownik

Nie przewidziano żadnego IDL do definiowania typów użytkownika.

Introspekcja XML-RPC

- Pojęcie introspekcji/refleksji powinno być znane z języka Java — mając obiekt można sprawdzić do jakiej klasy należy, jakie ma pola i jakich typów, wyliczyć jego metody itd.
- W odniesieniu do serwerów RPC introspekcja oznacza możliwość sprawdzenia jakie zdalne procedury są udostępnione, jakich typów argumentów oczekują i jakiego typu wyniki zwracają
- ... czasem dostępne są dodatkowe informacje, np. komentarz opisujący działanie procedury lub nazwy jej parametrów.
- Nieoficjalnym standardem introspekcji dla XML-RPC jest `http://xmlrpc-c.sourceforge.net/introspection.html`

JSON-RPC

- Zainspirowane przez XML-RPC (tak samo mówi o „metodzie” zamiast o „procedurze”), główna różnica: JSON zamiast XML.
- Zapytania i odpowiedzi przesyłane są jako słowniki JSON z określonymi w specyfikacji polami
- ... co za tym idzie argumenty i wyniki zdalnych procedur mogą mieć takie typy, jakie przewiduje JSON.
- Nie ma oficjalnego IDL.
- Oprócz HTTP POST dopuszczalne są inne sposoby przesyłania komunikatów JSON-RPC, np. przy pomocy gniazdek TCP.
- Wersja 1 z 2005: `https://www.jsonrpc.org/specification_v1`
- Wersja 2 z 2010: `https://www.jsonrpc.org/specification`

Kryptografia i kryptoanaliza

- Kryptografowie opracowują metody szyfrowania wiadomości (a także zapewnienia integralności, niezaprzeczalności, itd.).
- System szyfrowania w ujęciu matematycznym to dwie funkcje:

$$\text{E}(\text{tekst jawny, klucz szyfrujący}) = \text{szyfrogram}$$

$$\text{D}(\text{szyfrogram, klucz deszyfrujący}) = \text{tekst jawny}$$

- Atak *brute force* polega na przetestowaniu po kolei wszystkich możliwych kluczy deszyfrujących — zazwyczaj nierealne czasowo.
- Kryptoanalitycy próbują znaleźć słabość systemu, która pozwoli zmiejszyć liczbę testowanych kluczy, i/lub wykorzystać dodatkową wiedzę (np. znajomość pewnych fragmentów tekstu jawnego).
- W dzisiejszych czasach E i D są jawne.

Algorytmy szyfrowania symetrycznego

- Ten sam klucz jest używany do szyfrowania i deszyfrowania
- Problem: jak przesłać klucz, aby wróg go nie przechwycił?
- Długość klucza jest ustalona w definicji algorytmu.
- Większość współczesnych alg. jest blokowa z ustalonym n:

$$\text{E(blok n-bitowy, klucz)} = \text{blok n-bitowy}$$

- Trzeba umieć z nich korzystać — naiwne podzielenie wiadomości na bloki i szyfrowanie każdego oddzielnie ułatwi kryptoanalizę.
- DES: klucz 56 bitów, blok 64 bity.
- 3DES: klucz 168 bitów (efektywnie 112), blok 64 bity.
- AES: warianty z kluczami 128, 192 lub 256, blok 128 bitów.
- Powyższe trzy szyfry były/są oficjalnie zalecane przez NIST/NSA.
- Akceleracja sprzętowa AES we współczesnych procesorach.

Algorytmy sz. asymetrycznego z kluczem publicznym

- Dwa klucze związane w parę.
- Znajomość klucza sz. nie ułatwia odgadnięcia klucza desz.
- Klucz sz. można więc przesłać nadawcy wiadomości w dowolny sposób, nie martwiąc się że wróg go przechwyci.
- Można go nawet wydrukować w książce telefonicznej lub wywiesić na witrynie WWW, stąd „publiczny”.
- Klucz desz. czyli „prywatny” ma znać tylko odbiorca wiadomości.
- Najbezpieczniej gdy to on generuje parę kluczy.
- Dla dwukierunkowej komunikacji potrzebne są dwie pary kluczy.
- Skomplikowane matematycznie algorytmy, długość klucza nie ma bezpośredniego przełożenia na trudność złamania, porównywanie długości kluczy używanych w różnych alg. nie ma sensu.

Algorytm szyfrujący RSA (asym. z kluczem pub.)

- Bezpieczeństwo RSA zależy od trudności problemu rozkładu liczb na czynniki pierwsze.
- W latach 70 wystarczającą długością klucza RSA było 512 bitów, obecnie zalecane jest co najmniej 2048.
- Szyfrowana wiadomość teoretycznie może mieć tyle bitów ile klucz, w praktyce ma znacznie mniej.
- RSA jest powolne w porównaniu z algorytmami symetrycznymi, zazwyczaj więc używa się go razem z jakimś wybranym alg. sym.

Przykład dla RSA+AES, główne kroki:

- nadawca generuje losowy klucz AES
- szyfruje go kluczem publicznym RSA obiorcy i mu wysyła
- odbiorca swoim kluczem prywatnym RSA odkodowuje klucz AES
- nadawca wysyła wiadomości szyfrowane kluczem AES, odbiorca może wysyłać odpowiedzi używając tego samego klucza AES

Uzgadnianie klucza metodą Diffiego-Hellmana

- Bezpieczeństwo DH zależy od trudności problemu znajdywania dyskretnego logarytmu w grupie multiplikatywnej modulo.
- Wróg nie pozna klucza nawet jeśli przechwyci wszystkie wiadomości przesyłane podczas procedury generowania klucza
- ... ale może zastosować atak *man-in-the-middle*, bo w odróżnieniu od przykładu z poprz. slajdu brak weryfikacji tożsamości odbiorcy.

Główne kroki:

- strony uzgadniają niezbędne parametry obliczeń (w praktyce: jedna ze stron je generuje / miała wcześniej przygotowane i przesyła drugiej)
- każda ze stron generuje losową liczbę, wykonuje na niej obliczenia wg uzgodnionych parametrów, przesyła wynik partnerowi
- każda ze stron wykonuje drugą rundę obliczeń uwzględniającą odebrany wynik i uzgodnione parametry
- wynik drugiej rundy po obu stronach będzie taki sam – to jest klucz

Kryptograficzne funkcje skrótu (tzw. hasze)

- Podobnie jak alg. sum kontrolnych na wejściu dostają ciąg danych dowolnej długości i zwracają n-bitowy wynik (n jest ustalone).
- Każda zmiana danych wejściowych powoduje znaczącą i nieprzewidywalną zmianę wynikowego skrótu.
- Jednokierunkowe — znajomość skrótu nie mówi nic o oryginalnej wiadomości.
- Odporne na kolizje — nie da się wygenerować wiadomości mającej taki sam skrót jak inna inaczej niż metodą *brute force* (dla dostatecznie dużych n jest to więc praktycznie niemożliwe).
- MD5: skrót ma 128 bitów.
- SHA-1: 160 bitów.
- SHA-2: 224, 256, 384 lub 512 bitów.

Sygnatury (podpisy) kryptograficzne

- Używają kluczy publicznych i prywatnych.
- Pozwalają zweryfikować integralność i pochodzenie dokumentu.
- Autor oblicza skrót dokumentu jakimś wybranym alg. skrótu.
- Skrót i klucz prywatny służą do obliczenia podpisu.
- Podpis jest w jakiś sposób dołączany do oryginalnego dokumentu (odrębny plik, e-mail z załącznikiem MIME specjalnego typu, itp.).
- Odbiorca na własną rękę liczy skrót treści dokumentu, następnie wywołuje operację weryfikacji na skrócie, podpisie i kluczu pub. domniemanego autora.

Popularne alg. podpisu

RSA (jak widać nadaje się i do podpisów):

- klucze pub / priv jak przy szyfrowaniu
- funkcja skrótu kiedyś MD5, obecnie przeważnie SHA-256
- obliczenie podpisu — zaszyfrowanie skrótu kluczem prywatnym
- weryfikacja podpisu — zdekodowanie podpisu kluczem publicznym i sprawdzenie czy wynik równy własnoręcznie policzonemu skrótowi

DSA (znane też jako DSS):

- bazuje na problemie znajdowania logarytmu dyskretnego
- oryginalnie klucze DSA miały od 512 do 1024 bitów, obecnie zalecane jest 2048 lub 3072
- funkcją skrótu oryginalnie było SHA-1, obecnie używa się funkcji z rodziny SHA-2

HMAC

- HMAC to jeden z tzw. kodów uwierzytelniania wiadomości (MAC).
- Podobnie jak podpisy gwarantują integralność i autentyczność wiadomości, do której są dołączone.
- Zamiast pary kluczy pub / priv używa się pojedynczego klucza znanego obu stronom.
- Typowe zastosowanie: zabezpieczenie połączenia szyfrowanego alg. symetrycznym przed wrogimi próbami manipulacji.
- HMAC używa jakiegoś wybranego alg. skrótu: HMAC-MD5, HMAC-SHA-1, HMAC-SHA-256, itd.

Algorytmy oparte o krzywe eliptyczne

- Zamiast grup modulo używa się krzywych eliptycznych w ciałach liczb całkowitych (też nie musicie wiedzieć co to znaczy).
- Wymagają kluczy mniejszej długości niż poprzednia generacja alg.
- Odpowiednikiem DH jest ECDH.
- Odpowiednikiem DSA jest ECDSA.
- Obecnie zalecane są klucze ECDSA od 256 do 512 bitów.

Uwaga na boku: współcześnie używane alg. z kluczem publicznym są podatne na atak z użyciem komputera kwantowego; trwają badania nad potencjalnymi następcami algorytmów RSA, DSA i ECDSA.

Certyfikaty kryptograficzne — ogólna idea

Problem: skąd wziąć klucz publiczny osoby / serwera oferującego usługę sieciową, jeśli wcześniej nie miałem z nimi styczności?

- klucz publiczny osoby można by umieścić na jej stronie WWW, albo na serwerze pozwalającym wyszukiwać klucze wg adresów e-mail
- klucz publiczny serwera mógłby być wysyłany każdemu klientowi tuż po nawiązaniu przez niego połączenia

Problem: skąd pewność że to naprawdę ten właściwy klucz (wróg mógł się włamać na witrynę, podstawić swój własny serwer, itp.)?

- do klucza musi być dołączone poświadczenie własności wystawione przez kogoś, kto już wcześniej był mi znany
- ... i komu ufam, że przed wystawieniem poświadczenia dokładnie sprawdził, komu je wystawia

Certyfikaty X.509

- Standard X.509 definiuje strukturę certyfikatu, zasady ich wystawiania i awaryjnego odwoływania, hierarchię urzędów certyfikacyjnych (CA).
- Pojedynczy klocek wyrwany z rodziny standardów X.500 opracowanej przez ISO/OSI.
- Certyfikat zawiera:
  - klucz publiczny
  - okres ważności klucza (daty od–do)
  - dozwolone zastosowania klucza
  - nazwę właściciela klucza (subject)
  - nazwę urzędu poświadczającego autentyczność klucza (issuer)
  - pomocnicze metadane (np. typ i długość klucza)
  - podpis złożony kluczem prywatnym urzędu
- Typowe zastosowania: certyfikat serwera (szyfrowanie połączeń sieciowych), certyfikat osoby (podp. / sz. e-maili), certyfikat CA.
- Ciągi certyfikatów tworzące łańcuch zaufania.

Protokół SSL/TLS

- SSL (secure sockets layer) to rok 1995, Netscape Navigator 1.1
- przejęte przez IETF, rozwijane jako TLS (transport layer security)
- oryginalny cel: umożliwić działanie sklepom / bankom internetowym poprzez dostarczenie szyfrowanych tuneli dla HTTP
- przeglądarka musi być w stanie zweryfikować tożsamość serwera
- przesyłane pomiędzy przeglądarką a serwerem dane muszą być zabezpieczone przez podsłuchem / zmodyfikowaniem
- weryfikacja tożsamości użytkownika korzystającego z przeglądarki zazwyczaj jest robiona na wyższym poziomie (HTML-owy formularz logowania, sesje HTTP)

Główne kroki protokołu SSL/TLS

- klient otwiera połączenie
- klient i serwer negocjują wersję protokołu i alg. kryptograficzne
- serwer wysyła swój certyfikat i ewentualne certyfikaty pośrednich CA w łańcuszku zaufania
- klient sprawdza, czy *common name* właściciela certyfikatu pasuje do nazwy serwera, z którym próbował się połączyć
- klient weryfikuje autentyczność certyfikatu, łańcuch musi prowadzić do jednego ze znanych mu urzędów
- klient w dialogu z serwerem sprawdza, czy serwer ma prywatny klucz odpowiadający przedstawionemu certyfikatowi; klient i serwer ustalają jednorazowy klucz symetryczny
- rozpoczyna się transmisja strumienia właściwych danych, dzielonych na zaszyfrowane i opatrzone HMAC-ami pakiety

Dodatkowe uwagi do SSL/TLS

- trudno stworzyć system kryptograficzny bez żadnych usterek
- SSL/TLS mają w swej historii wiele błędów, zarówno na poziomie samego protokołu, jak i na poziomie implementacji
- diagnostyka witryn HTTPS (certyfikat, wspierane szyfry, obecność znanych usterek) np. `https://www.ssllabs.com/ssltest/`
- opcja w protokole: serwer może zażądać tzw. certyfikatu klienta, pozwalającego zweryfikować jego tożsamość (przydatne np. gdy zestawiamy tunel TLS pomiędzy dwoma bankami)
- inna opcja: klient może przy otwieraniu połączenia wysłać nazwę serwera (tzw. SNI) co ułatwia tworzenie wirtualnych witryn
- lista zaufanych CA wbudowana w przeglądarkę oraz to, czy wywiązują się one ze swych obowiązków są kluczowe dla bezpieczeństwa

Uwagi o urzędach certyfikacyjnych

- rola CA jest podobna do roli urzędu wydającego dowody osobiste
- potwierdzanie tożsamości na podstawie papierowych dokumentów jest czasochłonne, procedury więc rozluźniono
- certyfikaty typu *domain-validated* (DV) i *extended validation* (EV)
- pierwsze CA były monopolistami i kazały sobie słono płacić, współcześnie mamy darmowe Let's Encrypt
- wiele CA miało problemy (błędy w procedurach, włamania, penetracja przez agencje wywiadu)
- autorzy przeglądarek usunęli kilka CA ze swoich list, albo nałożyli ograniczenia na nazwy poświadczanych przez nie witryn
- zalecenia dotyczące certyfikatów patrz `https://cabforum.org/`

Alternatywny model zaufania: PGP

- podpisywanie i szyfrowanie listów elektronicznych
- zamiast hierarchii CA jest tzw. sieć zaufania *(web of trust)*
- pod certyfikatem z kluczem publicznym danej osoby może się podpisać dowolna liczba innych osób — robione jest to zazwyczaj na tzw. *key signing parties*
- klucze z podpisami są publikowane w Internecie i uaktualniane jeśli właściciel zdobędzie dodatkowe podpisy

Każdy użytkownik sam może zdecydować którym certyfikatom ufa:

- tylko tym, które dostałem bezpośrednio od właścicieli
- ... i tym, pod którymi podpisali się Adam lub Barbara, bo tę dwójkę znam i wiem, że sprawdzają komu podpisują
- ... i tym, pod którymi podpisali się ludzie posiadający certyfikaty podpisane przez Adama lub Barbarę
- itd. z ewentualnymi wariacjami (np. wymagane są dwa podpisy)

Alternatywny model zaufania: SSH

- szyfrowanie zdalnych sesji terminalowych
- serwer wysyła swój klucz po nawiązaniu połączenia
- klient ma podręczny spis serwerów, z którymi już się kiedyś łączył i ich kluczy
- jeśli klucz się zgadza kontynuuj
- jeśli w spisie nie ma tego serwera daj znać użytkownikowi, że łączy się po raz pierwszy i poproś o zaakceptowanie klucza (użytkownik może wtedy np. zadzwonić do administratora serwera i tą drogą zweryfikować klucz)
- jeśli klucz się nie zgadza podnieś alarm
- podgląd negocjacji między klientem a serwerem:

```
ssh -v -v spk-ssh.if.uj.edu.pl
```

Biblioteka OpenSSL

- najpowszechniej używana uniksowa biblioteka SSL/TLS
- alternatywy: GnuTLS, NSS opracowana na potrzeby Firefoksa, LibreSSL czyli fork OpenSSL rozwijany w ramach OpenBSD itd.
- biblioteka, a nie część jądra systemu operacyjnego, nie można więc „stworzyć gniazdka TLS”
- OpenSSL ma swoje własne typy danych i funkcje:

```
SSL * SSL_new(SSL_CTX * ctx);
int SSL_set_fd(SSL * ssl, int fd);
int SSL_connect(SSL * ssl);
int SSL_accept(SSL * ssl);
int SSL_read(SSL * ssl, void * buf, int num);
int SSL_write(SSL * ssl, const void * buf, int num);
void SSL_free(SSL * ssl);
```

- jak widać, `SSL *` pełni rolę analogiczną do `FILE *`

Biblioteka OpenSSL

- kontekst definiuje parametry połączenia
- dopuszczalne wersje protokołu SSL/TLS, szyfry, itp.
- dla klienta: lista zaufanych urzędów certyfikacyjnych
- dla serwera: jego certyfikat i klucz prywatny
- domyślny zbiór zaufanych CA jest w `/etc/ssl/certs/`
- pliki w `/etc/ssl/` są zazwyczaj w tekstowym formacie PEM
- PEM wprowadzono bo standard X.509 oryginalnie zapisywał klucze, certyfikaty, itd. w binarnym formacie DER, a w dawnych czasach nie można było wysłać w liście binarnego załącznika
- plik PEM to plik DER zakodowany w Base64 (każde trzy bajty są zastępowane czterema drukowalnymi znakami ASCII) z dodanymi górną i dolną linią ograniczającą

Polecenie openssl

- ręczne łączenie się z serwerami używającymi SSL/TLS:
  openssl s_client -crlf -showcerts -connect mail.uj.edu.pl:465
  (porównaj z ncat -C -v –ssl mail.uj.edu.pl 465)
- generowanie i zarządzanie kluczami, np.
  openssl genrsa -out witryna.key 2048
- generowanie tzw. CSR, czyli prośby o podpisanie certyfikatu:
  openssl req -new -key witryna.key -out witryna.csr
  (nazwę witryny podaje się z klawiatury podczas generowania)
- generowanie tzw. *self-signed certificates*:
  openssl req -new -x509 -days 365 -key witryna.key -out witryna.crt
- podgląd zawartości certyfikatu w formie czytelnej dla ludzi:
  openssl x509 -in witryna.crt -noout -text
- odpowiadanie na przychodzące połączenia:
  openssl s_server -cert witryna.crt -key witryna.key -port 1234
  (por. ncat –ssl-cert witryna.crt –ssl-key witryna.key -l 1234)

Obsługa SSL/TLS w Pythonie

Python ukrywa przed programistą dużą część złożoności OpenSSL.

```
>>> import socket, ssl

>>> s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
>>> s.connect( ('smtp.gmail.com', 465) )
>>> tls_s = ssl.wrap_socket(s)
>>> tls_s.recv(1024)
b'220 smtp.gmail.com ESMTP g19sm5951224lfh.32 - gsmtp\r\n'
>>> tls_s.close()

>>> s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
>>> tls_s = ssl.wrap_socket(s)
>>> tls_s.connect( ('smtp.gmail.com', 465) )
>>> tls_s.recv(1024)
b'220 smtp.gmail.com ESMTP h14sm6246122ljg.10 - gsmtp\r\n'
>>> tls_s.close()
```

Obsługa SSL/TLS w Pythonie

Trzeba uważać na szczegóły, w przeciwnym razie może się okazać że nie jesteśmy tak bezpieczni jak sobie wyobrażaliśmy. Spróbujmy połączyć się z testowym serwerem TLS używającym certyfikatu podpisanego przez niego samego:

```
>>> s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
>>> tls_s = ssl.wrap_socket(s)
>>> tls_s.connect( ('127.0.0.1', 1234) )
>>> tls_s.close()

>>> s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
>>> tls_s = ssl.wrap_socket(s, cert_reqs=ssl.CERT_REQUIRED,
        ca_certs='/etc/ssl/certs/ca-certificates.crt')
>>> tls_s.connect( ('127.0.0.1', 1234) )
ssl.SSLError: ... certificate verify failed
```

Obsługa SSL/TLS w Pythonie

```
>>> s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
>>> tls_s = ssl.wrap_socket(s, server_side=True,
        certfile='witryna.crt', keyfile='witryna.key')
>>> tls_s.bind( ('0.0.0.0', 1234) )
>>> tls_s.listen()
>>> while True:
...     cl_sock, cl_addr = tls_s.accept()
...     cl_sock.send(b'Hello!\r\n')
...     cl_sock.close()
```

Obsługa SSL/TLS w Javie

Do Javy począwszy od wersji 1.4 standardowo dołączany jest moduł JSSE obsługujący połączenia SSL/TLS. Jest zaimpl. w całości w Javie i dzięki temu przenośny. Najważniejsze klasy i interfejsy:

- `SSLSocket`
- `SSLSocketFactory`
- `SSLServerSocket`
- `SSLServerSocketFactory`
- `SSLContext`
- `X509TrustManager`
- `X509KeyManager`

Plik `$JAVA_HOME/lib/security/cacerts` zawiera domyślny zbiór certyfikatów należących do zaufanych urzędów.

Obsługa SSL/TLS w Javie

```
SSLSocketFactory fac = (SSLSocketFactory)
        SSLSocketFactory.getDefault();
SSLSocket sock = (SSLSocket)
        fac.createSocket("www.uj.edu.pl", 443);
InputStream in = sock.getInputStream();
OutputStream out = sock.getOutputStream();
out.write(
        "HEAD / HTTP/1.0\r\n\r\n".getBytes("US-ASCII") );
byte[] buf = new byte[1024];
int len;
while (0 < (len = in.read(buf))) {
    System.out.print(new String(buf, 0, len, "US-ASCII"));
}
sock.close();   // automatycznie zamknie też in oraz out
```

Serializacja jednego obiektu

Na pierwszy rzut oka serializowanie obiektów może się wydawać podobne do serializowania struktur – zserializuj po kolei wszystkie pola i gotowe.

Po drugim rzucie oka widać problem: jeśli zamiast obiektu oczekiwanej klasy pojawi się obiekt klasy potomnej, to co wtedy? Dwa podejścia:

- rzutujemy obiekt do typu klasy bazowej, zapisujemy pola z bazowej
- zapisujemy do strumienia bajtów rzeczywistą klasę obiektu (np. jako łańcuch), a potem wszystkie pola obiektu (i te odziedziczone z klasy bazowej, i te dodane w pochodnej)

Pierwsze podejście przy deserializacji da nam obiekt klasy bazowej, a drugie – obiekt klasy pochodnej. Praktycznie zawsze korzysta się więc z tego drugiego.

Serializacja wielu obiektów

Obiekty można przy pomocy wskaźników/referencji łączyć w sieci (czy też bardziej formalnie: w grafy). Naiwny algorytm serializacji, rekrusywnie podążający za wskaźnikami zapisanymi w polach przetwarzanych obiektów, nie poradzi sobie jeśli do jakiegoś obiektu można dojść więcej niż jedną ścieżką – taki obiekt zostanie wielokrotnie zapisany. A jeśli w grafie są cykle, to algorytm wpadnie w pętlę nieskończoną.

Konieczne jest użycie algorytmu zapamiętującego, które obiekty już odwiedził. W przypadku natknięcia się na wskaźnik do już widzianego obiektu ten wskaźnik musi zostać przetworzony w sposób szczególny – do strumienia zapisywany jest nie obiekt, na który pokazuje, lecz specjalna wartość. Mówi ona „tu podczas deserializacji należy wstawić adres obiektu, który został ze strumienia wcześniej odczytany”. Oczywiście potrzebny jest jakiś sposób określania, o który z poprzednich obiektów chodzi, można np. je numerować 1, 2, 3 itd.

Inna specjalna wartość reprezentuje puste referencje (`null`).

Serializacja w C++ i Javie

C++ nie ma wbudowanej w język obsługi serializacji. Można ją zaimplementować samodzielnie, przy czym należy wcześniej uważnie przeczytać sekcję Serialization and Unserialization w C++ FAQ.

Proszę ją zresztą przeczytać nawet jeśli nie programujecie w C++, dyskusja tam zawarta jest doskonałym uzupełnieniem poprzednich slajdów.

Alternatywnie, można skorzystać z którejś z kilku dostępnych bibliotek: Boost::Serialization, cereal, itd.

Java wspiera serializację, patrz interfejs `Serializable` i rzeczy z nim związane. Szczegółowa techniczna specyfikacja jest dostępna tutaj.

Zdalne obiekty

Zdalny obiekt rezyduje na innym komputerze, ale poza tym chcielibyśmy móc z niego korzystać tak samo, jak się korzysta ze zwykłych, lokalnych obiektów. Chcemy mieć:

- zmienne zawierające referencje do zdalnych obiektów
- możliwość kopiowania tych referencji, porównywania ich itp.
- możliwość wywoływania metod w zdalnym obiekcie
- dostęp do pól w zdalnym obiekcie nie jest konieczny (możemy się umówić, że wszystkie pola są prywatne i zawsze korzystać z metod dostępowych)

Przykładami standardów definiujących pojęcie zdalnej metody są, w kolejności chronologicznej, CORBA, Java RMI oraz SOAP.

CORBA i SOAP są wieloplatformowe i nawet mają bindingi dla języków nie-obiektowych. Java RMI jest ściśle związany z Javą i przez to znacznie od nich prostszy.

Java RMI (Remote Method Invocation)

- Listę metod udostępnianych przez zdalny obiekt definiuje się po prostu jako interfejs Javy. Nie ma odrębnego IDL.
- Wszystkie metody zdalnych obiektów mogą zgłaszać wyjątki sygnalizujące wystąpienie problemów komunikacyjnych.
- Na podstawie skompilowanego interfejsu generowane są pomocnicze *stub class* oraz *skeleton class*. W pierwszej wersji RMI programista robił to ręcznie poleceniem `rmic`. Późniejsze wersje używają refleksji i generują namiastki oraz szkielety w locie.
- Programista odpowiedzialny za serwer pisze klasę implementującą zdalny interfejs, następnie tworzy i przekazuje bibliotece RMI obiekt tej klasy. Ona opakowuje go w szkielet, udostępnia w sieci i tworzy namiastkę reprezentującą ten obiekt.
- Tę namiastkę trzeba udostępnić potencjalnym klientom. Zazwyczaj publikuje się ją pod uzgodnioną wcześniej nazwą w katalogu obiektów utrzymywanym przez pomocniczego demona `rmiregistry`.

Java RMI: interfejs

Przykładem, tak jak poprzednio, jest dodawanie ciągów liczb.

W Javie ciągi wartości można przechowywać w tablicach, ale również w obiektach-pojemnikach. Niech więc będą dwie wersje metody:

```
public interface Sumator extends java.rmi.Remote
{
  public int dodaj(int[] liczby)
    throws SumatorException, java.rmi.RemoteException;
  public int dodaj(Iterable<Integer> liczby)
    throws SumatorException, java.rmi.RemoteException;
}

// Do sygnalizowania przepełnienia przy dodawaniu:
public class SumatorException extends Exception
{ }
```

Java RMI: implementacja metod obiektu

```
public class SumatorImpl implements Sumator
{
  public int dodaj(int[] liczby) throws SumatorException
  {
    int wynik = 0;
    try {
      for (int x : liczby) {
        wynik = Math.addExact(wynik, x);
      }
    } catch (ArithmeticException ex) {
      throw new SumatorException();
    }
    return wynik;
  }
  ..... // druga wersja metody analogicznie
}
```

Java RMI: uruchamianie serwera

```
public static void main(String[] args)
  throws Exception
{
  SumatorImpl sumator = new SumatorImpl();
  Sumator namiastka = (Sumator)
      java.rmi.server.UnicastRemoteObject
      .exportObject(sumator, 0);

  String adres = "//localhost/sumator";
  java.rmi.Naming.rebind(adres, namiastka);
}
```

Powyższy kod nie obsługuje wyjątków, które mogą wystąpić przy próbie wyeksportowania obiektu-sumatora lub wpisania go do rejestru.

Java RMI: zarządzanie rejestrem

Klasa `Naming` ma następujące statyczne metody:

- `void bind(String address, Remote obj)`
- `void rebind(String address, Remote obj)`
- `Remote lookup(String address)`
- `void unbind(String address)`

Format używanych adresów wzorowany jest na formacie URL-i:

//host:port/name

Nazwa hosta i numer portu są opcjonalne, ich domyślne wartości to `localhost` oraz 1099. Nazwa obiektu może być dowolnym niepustym ciągiem znaków.

Uwaga: host i port opisują lokalizację rejestru, a nie zdalnego obiektu!

Java RMI: klient używający tablicy

```
public static void main(String[] args) throws Exception
{
  var adres = "/sumator";
  var sumator = (Sumator) java.rmi.Naming.lookup(adres);

  int[] tablica = new int[] { 2, 2 };
  int wynik = sumator.dodaj(tablica);

  tablica[0] = Integer.MAX_VALUE;
  tablica[1] = Integer.MAX_VALUE;
  try {
    wynik = sumator.dodaj(tablica);
  } catch (SumatorException ex) {
    System.out.println("zbyt wielka suma");
  }
}
```

Java RMI: protokół transportowy

RMI domyślnie korzysta z klas `Socket` i `ServerSocket` z pakietu `java.net`. Standardowym transportem jest więc TCP/IP.

Klient z poprzedniego slajdu otworzy dwa połączenia:

- do rejestru na 127.0.0.1:1099
- do obiektu-sumatora na taki adres IP i numer portu, jakie będą w namiastce z tego rejestru pobranej

RMI pozwala na podanie naszej własnej fabryki gniazdek klienckich i serwerowych. Pozwala to np. zastąpić zwykłe połączenia TCP/IP szyfrowanymi połączeniami TLS.

Wewnątrz pojedynczej maszyny wirtualnej Javy na jednym nasłuchującym gniazdku może być dostępnych kilka zdalnych obiektów. Jest to możliwe, bo podczas eksportowania obiekty otrzymują indywidualne identyfikatory.

Java RMI: zarządzanie rejestrem c.d.

Wśród wyjątków, które mogą zgłaszać metody z klasy `Naming` jest `RemoteException`. Sugeruje to, że rejestr jest zaimplementowany jako zdalny obiekt i rzeczywiście tak jest.

Interfejs `java.rmi.registry.Registry` specyfikuje metody rejestru. Też są w nim `bind`, `rebind`, `lookup` i `unbind`, ale jako argumenty podaje się tylko nazwy obiektów, a nie pełne adresy.

Metody klasy `Naming` rozkładają podany im adres na części, tworzą namiastkę `Registry` odwołującą się do wskazanego hosta i portu, poprzez nią żądają wykonania akcji na wskazanej nazwie obiektu.

Java RMI: zarządzanie rejestrem c.d.

Aby nie pojawił się problem jajka i kury (aby wywołać metodę rejestru trzeba mieć jego namiastkę, a namiastki są pobierane z rejestru) w klasie `java.rmi.registry.LocateRegistry` są dwie statyczne metody:

- `Registry getRegistry(String host, int port)`

  Konstruuje namiastkę wskazującą na rejestr dostępny pod podanym adresem TCP/IP.

- `Registry createRegistry(int port)`

  Tworzy i eksportuje nowy rejestr nasłuchujący na podanym porcie oraz tworzy i zwraca namiastkę na niego wskazującą.

Pierwsza metoda jest dla klientów. Z drugiej korzysta `rmiregistry` i te serwery, które chcą być niezależne od zewnętrznego rejestru.

Java RMI: przykład użycia metody getRegistry

Zazwyczaj nie ma powodu, aby w kliencie korzystać z `getRegistry` (klasa `Naming` jest wygodniejsza), ale w razie potrzeby robiłoby się to mniej więcej tak:

```
public static void main(String[] args) throws Exception
{
  String host = (args.length < 1) ? "localhost" : args[0];
  int port = (args.length < 2) ?
    Registry.REGISTRY_PORT : Integer.parseInt(args[1]);
  String nazwa = (args.length < 3) ? "sumator" : args[2];

  Registry rejestr = LocateRegistry.getRegistry(host, port);
  Sumator sumator = (Sumator) rejestr.lookup(nazwa);

  .....
}
```

Java RMI: serwer z własnym rejestrem

```
import static
    java.rmi.registry.LocateRegistry.createRegistry;
import static
    java.rmi.server.UnicastRemoteObject.exportObject;

public static void main(String[] args) throws Exception
{
  int port = (args.length < 1) ? 2020
        : Integer.parseInt(args[0]);
  String nazwa = (args.length < 2) ? "sumator" : args[1];

  var rejestr = createRegistry(port);
  var sumator = new SumatorImpl();
  var obj = exportObject(sumator, port);
  rejestr.bind(nazwa, obj);
}
```

Java RMI: serwer z własnym rejestrem

Brak centralnego rejestru ma i zalety, i wady.

Plusy:

- nie trzeba uruchamiać pomocniczej infrastruktury
- klientom podaje się adres IP i port TCP zdalnego obiektu, czyli to samo co w przypadku innych internetowych usług

Minusy:

- zmiana komputera, na którym działa zdalny obiekt pociąga za sobą konieczność uaktualnienia konfiguracji wszystkich klientów
- brak centralnego rejestru utrudnia implementację zaawansowanych scenariuszy migracji oraz uaktualniania oprogramowania

Java RMI: klient używający pojemników

```
public static void main(String[] args) throws Exception
{
  .....

  var wektor = new java.util.Vector<Integer>();
  wektor.add(12);
  wektor.add(24);
  wektor.add(8);
  System.out.println(sumator.dodaj(wektor));

  var lista = new MojaLista<Integer>();
  lista.add(12);
  lista.add(24);
  lista.add(8);
  System.out.println(sumator.dodaj(lista));
}
```

Java RMI: dystrybucja niezbędnych klas

Pliki `Sumator.class` i `SumatorException.class` muszą być dostępne nie tylko dla serwera i klientów, ale również dla JVM, w której działa rejestr. Jest to konieczne by szkielet obiektu-rejestru mógł zdeserializować przesłaną mu namiastkę (patrz typ parametrów w `bind` i `rebind`).

W przykładzie metoda `dodaj` jako argument dostaje obiekt-pojemnik. Jeśli zamiast jednego z pojemników z bibl. standardowej próbujemy przekazać coś, co sami zaimplementowaliśmy, to tę własną klasę trzeba serwerowi udostępnić. Najprostszym rozwiązaniem jest umieszczenie niezbędnych plików `*.class` w odpowiednim miejscu na dysku serwera.

Możliwe przeszkody: administrator serwera ignoruje nasze prośby; kolizja nazw klas, bo w różnych programach klienckich zaimpl. pojemniki tak samo się nazywające.

Java RMI: dystrybucja niezbędnych klas

Dawne przeglądarki obsługiwały applety Javy (programy osadzane na stronach HTML). Pozostał po nich mechanizm pobierania kodu klas na żądanie. RMI potrafi z niego korzystać, choć nie jest to zalecane, bo złośliwy klient może próbować podsunąć serwerowi klasę coś niecnego robiącą. Niezbędne kroki:

- na komputerze klienckim uruchom serwer HTTP, który udostępnia `MojaLista.class` oraz jej klasy pomocnicze
- klienckiej JVM podaj jako opcję URL powyższego serwera (dzięki temu serializowane obiekty obok nazwy ich klasy będą miały adnotację mówiącą skąd tę klasę można ściągnąć)
- serwerowej JVM włącz pobieranie klas ze zdalnych źródeł oraz określ politykę bezpieczeństwa

Java RMI: końcowe różności

- W nadchodzących wersjach Javy może zostać usunięta obsługa menedżerów bezpieczeństwa i ich polityk. RMI może wtedy stracić możliwość pobierania definicji klas przez sieć.
- Jeśli w systemie ma być wiele rzadko wykorzystywanych zdalnych obiektów, to dobrze byłoby je tworzyć tylko na ten okres, gdy są potrzebne. Mechanizm RMI Activation był przeznaczony dla takich scenariuszy, ale nie był popularny i go usunięto.
- Gdy zdalne obiekty mogą być dostępne za pośrednictwem różnych protokołów (np. RMI oraz CORBA), to adresy RMI zapisuje się używając schematu *rmi*. Są wtedy poprawnymi URL-ami.

  Przykład: `rmi://registry.example.org/sumator`

Kilka słów o CORBA

- Pełna nazwa: Common Object Request Broker Architecture.
- Wieloplatformowy standard rozwijany przez konsorcjum OMG, czyli Object Management Group.
- OMG IDL (typy danych, interfejsy i metody, moduły).
- Do pisania aplikacji oprócz kompilatora IDL potrzebna jest biblioteka implementująca ORB (Object Request Broker) i powiązane z nim specyfikacje takie jak IIOP (Internet InterORB Protocol).
- Istnieją standardy tłumaczenia deklaracji z plików IDL na konstrukcje dostępne w poszczególnych językach programowania (Sun RPCL ich nie miał, każda biblioteka robiła po swojemu).
- CORBAservices i CORBAfacilities, czyli standardowe usługi często potrzebne typowym aplikacjom.
- Zdecydowanie bardziej zaawansowane i skomplikowane niż rozwiązania wcześniej na wykładzie omawiane.

Kilka słów o SOAP

- Oryginalnie miał to być Simple Object Access Protocol
- ... a potem zaczął nad nim pracować komitet.
- Podstawowym pojęciem jest wiadomość (ang. *message*).
- Niby XML, ale może być transmitowana w specjalnych binarnych formatach (bardziej efektywne niż tekstowy XML)
- ... i z możliwością dodawania załączników MIME.
- Oprócz HTTP POST są przewidziane inne formy transportu wiadomości, w tym nawet wewnątrz e-maili.
- Pomiędzy nadawcą a finalnym odbiorcą może być kilka serwerów pośredniczących, które przetwarzają / modyfikują wiadomość.
- „Przesłanie wiadomości od do" jest podstawowym prymitywem komunikacyjnym, z którego budowane są bardziej złożone scenariusze (np. wywołanie zdalnej procedury i zwrócenie wyniku).

Message-oriented middleware

Oprogramowanie typu message-oriented middleware (MOM) wspiera tworzenie rozproszonych systemów, w których komunikacja pomiędzy komponentami oparta jest o przesyłanie komunikatów. Nadawca wysyła wiadomość nie bezpośrednio do odbiorcy, lecz do kolejki utrzymywanej przez brokera. Broker albo od razu przesyła komunikat dalej, do jednego lub więcej odbiorców związanych z tą kolejką, albo go magazynuje do chwili gdy któryś odbiorca o niego poprosi.

Wprowadzenie brokera-pośrednika oraz kolejek pozwala na:

- asynchroniczne działanie nadawców i odbiorców
- implementację różnych wzorców komunikacyjnych, takich jak klient-serwer, load balancer, publicysta-subskrybenci czy potok push-pull poprzez odpowiedni dobór typu kolejki
- restart/migrację komponentu bez wpływania na pozostałe

ZeroMQ

ZeroMQ jest nietypowym MOM, bo nie korzysta z brokerów. Zamiast tego biblioteka tworzy podczas uruchamiania programu dodatkowe wątki, zajmujące się przesyłaniem wiadomości. Można powiedzieć, że każdy program ma swojego prywatnego, wewnętrznego brokera.

W ZeroMQ nie ma kolejek (a przynajmniej programista ich nie widzi). Są tylko gniazdka pozwalające łączyć się z innymi programami (proszę nie sugerować się zbieżnością nazw, to nie to samo co gniazdka POSIX). Typ wybrany podczas tworzenia gniazdka określa, jaki wzorzec komunikacyjny będzie realizowany. Aby pomiędzy dwoma gniazdkami dało się stworzyć połączenie, ich typy muszą być ze sobą kompatybilne.

Typy gniazdek ZeroMQ

publicysta-subskrybent
: `ZMQ_PUB` / `ZMQ_SUB` (oraz `ZMQ_XSUB` i `ZMQ_XPUB`)

klient-serwer
: `ZMQ_REQ` / `ZMQ_REP` (oraz `ZMQ_ROUTER` i `ZMQ_DEALER`)

jednokierunkowy potok
: `ZMQ_PUSH` / `ZMQ_PULL`

zwykłe połączenie TCP
: `ZMQ_STREAM`

Wszystkie typy pozwalają na podłączenie gniazdka do więcej niż jednego rozmówcy. Dodatkowe typy wzięte w nawiasy są używane do impl. programów pośredniczących w przesyłaniu wiadomości.

Przykład klienta

```
void * ctx = zmq_ctx_new();
void * sock = zmq_socket(ctx, ZMQ_REQ);
zmq_connect(sock, "tcp://127.0.0.1:2021");

zmq_send(sock, "ping", 4, 0);

char buf[16];
n = zmq_recv(sock, buf, sizeof(buf), 0);

zmq_close(sock);
zmq_ctx_term(ctx);
```

Uwaga: w przykładzie brak obsługi błędów.

Klient może zostać uruchomiony przed serwerem.

Przykład serwera

```
void * ctx = zmq_ctx_new();
void * sock = zmq_socket(ctx, ZMQ_REP);
zmq_bind(sock, "tcp://*:2021");

char buf[16];
n = zmq_recv(sock, buf, sizeof(buf), 0);

if ((n == 4) && (memcmp(buf, "ping", 4) == 0)) {
    zmq_send(sock, "pong", 4, 0);
} else {
    zmq_send(sock, "ERROR", 5, 0);
}

zmq_close(sock);
zmq_ctx_term(ctx);
```

Wiadomości wieloczęściowe

W ZeroMQ komunikat może składać się z kilku oddzielnych części.
Wysyłkę robi się tak:

```
zmq_send(sock, "Ala", 3, ZMQ_SNDMORE);
zmq_send(sock, "ma", 2, ZMQ_SNDMORE);
zmq_send(sock, "kota", 4, 0);
```

Odbiorca będzie musiał trzy razy wywołać `zmq_recv()`.

Jeśli nie wiadomo z góry, że będą trzy części, to trzeba sprawdzać czy w metadanych odebranej wiadomości jest flaga „będzie więcej”:

```
zmq_msg_t msg;
do {
    zmq_msg_init(&msg);
    zmq_msg_recv(&msg, sock, 0);
    // zrób coś z danymi w msg
    zmq_msg_close(&msg);
} while (zmq_msg_more(&msg));
```

Łączność z użyciem zwykłego TCP

Gniazdko `STREAM` może i odbierać, i nawiązywać połączenia TCP. Nawet gdy jest otwartych wiele równoczesnych połączeń TCP, na poziomie kodu aplikacji dalej jest tylko jedno gniazdko ZeroMQ. Potrzebny jest więc mechanizm pozwalający wskazywać, do którego z gniazdek TCP chcemy dane zapisać, oraz z którego pochodzą właśnie odczytane dane.

Twórcy ZeroMQ przyjęli, że każda odczytana/zapisywana wiadomość będzie miała dwie części: w pierwszej jest identyfikator gniazdka TCP (arbitralny ciąg bajtów), w drugiej właściwe dane.

Następne slajdy pokazują, jak w ZeroMQ zaimplementować odpowiednik przykładu `rot13_server.c`. Proszę zwrócić uwagę, że wszystkie rzeczy związane z korzystaniem z `select()`, `poll()` czy też jakiegoś innego mechanizmu jądra odpowiedzialnego za multipleksowanie deskryptorów są obsługiwane przez bibliotekę ZeroMQ.

Równoległa obsługa wielu klientów TCP

```
#include <vector>
typedef std::vector<char> rid_t;

void * sock = zmq_socket(ctx, ZMQ_STREAM);
zmq_bind(sock, "tcp://*:2021");
while (true) {
    // Pierwsza część z routing id.
    zmq_msg_init(&msg);
    n = zmq_msg_recv(&msg, sock, 0);
    char * data = (char *) zmq_msg_data(&msg);
    rid_t rid(data, data + n);
    zmq_msg_close(&msg);

⋮
```

Równoległa obsługa wielu klientów TCP

```
⋮

  // Druga część z danymi do zaszyfrowania.
  zmq_msg_init(&msg);
  n = zmq_msg_recv(&msg, sock, 0);
  if (n > 0) {
      data = (char *) zmq_msg_data(&msg);
      rot13(data, n);
      zmq_send(sock, rid.data(), rid.size(), ZMQ_SNDMORE);
      zmq_msg_send(&msg, sock, 0);
  }
  zmq_msg_close(&msg);
}
```

Przypomnienie: w przykładach nie ma obsługi błędów.

Internet Protocol wersja 6

- Formalna specyfikacja patrz RFC 2460.
- IPv4 miało 32-bitowe adresy (4 bajty)
- ... a IPv6 wydłużyło je do 128-bitowych (16 bajtów).
- Tekstowa postać adresów:

  0102:0304:0506:0708:090A:0B0C:0D0E:0F10

  102:300:: = 0102:0300:0000:0000:0000:0000:0000:0000
- Zamiast ARP jest Neighbor Discovery Protocol (NDP); potrafi on również wykrywać routery i zastąpić DHCP.
- Oprócz globalnych adresów są też adresy lokalne, tzw. link-local, zaczynające się od FE80::/10
- Adres loopback to ::1 (odpowiednik 127.0.0.1).

Implementacje IPv6 w jądrach systemów operacyjnych

- Najczęściej jądro ma niezależne moduły IPv4 i IPv6, interfejsy mają przypisane i takie, i takie adresy — jest to tzw. podejście dual stack.
- API gniazdek rozszerzono tak, aby mogły obsługiwać połączenia IPv6 (szczegółowy opis patrz RFC 3493).
- `struct sockaddr_in6`, `struct sockaddr_storage`
- `inet_pton()`, `inet_ntop()` — z/do liczbowej postaci tekstowej
- `getaddrinfo()`, `getnameinfo()` — z/do nazwy w DNS
- W okresie przejściowym serwery będą musiały oferować swe usługi i przez IPv4, i przez IPv6
- ... dla wygody wprowadzono więc możliwość obsługi połączeń IPv4 przez gniazdka IPv6 używające tzw. IPv4-mapped addresses.

Adresy typu IPv4-mapped

- Podsieć ::FFFF:0:0/96 odpowiada adresom IPv4:

  $$1.2.3.4 \quad \longleftrightarrow \quad ::FFFF:0102:0304$$

  (przeważnie zapisywane jako ::FFFF:1.2.3.4)
- Zarezerwowana tylko do tego celu, adresy z tego przedziału nie pojawiają się w nagłówkach pakietów IPv6.
- Nasłuchujące gniazdko IPv6 może odebrać połączenie IPv4; jeśli aplikacja spyta o adres klienta, to dostanie wartość IPv4-mapped.
- Gniazdko IPv6 może też zainicjować połączenie IPv4, jeśli jako argument dla `connect()` podany zostanie adres IPv4-mapped.
- Ten hybrydowy tryb pracy jest domyślny. W razie potrzeby można go wyłączyć ustawiając gniazdku opcję `IPV6_V6ONLY`.
- `getaddrinfo()` potrafi konwertować pobierane z DNS adresy IPv4 do postaci IPv4-mapped.