

Programowanie aplikacji sieciowych

Skrypt
(Wersja nieskończona)

DAMIAN RUSINEK

5 czerwca 2017

Spis treści

1 Modele komunikacji	1
1.1 Model OSI	1
1.2 Model TCP/IP	3
2 Protokoły komunikacyjne	7
2.1 Podgląd pakietów	7
2.1.1 Podsłuchiwanie pakietów	7
2.1.2 Filtry	8
2.2 ARP	11
2.2.1 Format ARP	11
2.3 IP	13
2.3.1 Klasy IP	14
2.3.2 Trasowanie	15
2.3.3 Format IP	16
2.3.4 IPSec	19
2.4 TCP	19
2.4.1 Format TCP	20
2.4.2 Połączenie TCP	21
2.4.3 Stan połączenia	22
2.4.4 Przykład podsłuchanego pakietu TCP	23
2.5 UDP	23
2.5.1 Format UDP	24
2.5.2 Przykład podsłuchanego pakietu UDP	24
2.6 DNS	25
2.6.1 Format DNS	26
2.6.2 Przykład podsłuchanej komunikacji DNS	28
3 Architektury sieci	31
3.0.1 Klient-Serwer	31
3.0.2 Peer-2-Peer	32
3.1 Metody komunikacji	34
3.1.1 Request-response	35
3.1.2 Publish-subscribe	35

4 Programowanie aplikacji sieciowych	37
4.1 Gniazda	37
4.1.1 Metody	39
4.1.2 Metody pomocnicze	51
4.1.3 Prosty klient DNS	53
4.1.4 Prosta aplikacja strumieniowa	56
4.1.5 Prosta aplikacja bezpołączeniowa (UDP)	65
4.1.6 Protokół IP w wersji 6	71
4.2 Protokół poczty	71
4.2.1 Wysyłanie wiadomości e-mail	71
4.2.2 Pobieranie wiadomości e-mail	79
4.3 Protokół HTTP	87
4.3.1 Żądanie HTTP	87
4.3.2 Odpowiedź HTTP	88
4.3.3 Nagłówki	89
4.3.4 Kody odpowiedzi	96
4.3.5 Prosty klient HTTP	100
4.3.6 API po HTTP	103
4.3.7 Websocket	110
4.4 Analiza protokołu	116
4.4.1 KNX	116
4.4.2 Protokół KNXnet/IP	119
4.4.3 Analiza ruchu	126
4.4.4 Budowa klienta KNXnet/IP	131
4.4.5 Krótka wzmianka o bezpieczeństwie	139
4.5 Własny protokół	141
4.5.1 Kółko i krzyżyk	142
5 Mechanizmy zaawansowane	149
5.1 Zaawansowany chat	149
5.1.1 Wielowątkowość	151
5.1.2 Protokół chatu	152
5.1.3 Synchronizacja wątków	154
5.1.4 Klient Chat	158
5.2 Bezpieczne gniazda TLS/SSL	160
5.2.1 Klient HTTPS	161
5.2.2 Szyfrowany czat	164
5.2.3 Tunelowanie SSH	168
5.3 Serwery zdarzeniowe	173
5.3.1 Chat zdarzeniowy	174
5.3.2 Chat zdarzeniony w asyncio	178

1

Modele komunikacji

1.1 Model OSI

Rozpoczęcie programowanie aplikacji sieciowych, czy to aplikacji typu klient, czy serwer, wymaga zapoznania się z podstawowymi zagadnieniami dotyczącymi sieci.

Pierwszym z zagadnień jest model OSI (ang. Open Systems Interconnection), czyli standard zdefiniowany przez organizację ISO (International Organization for Standardization), opisujący strukturę komunikacji sieciowej. Model ISO OSI RM (ang. Reference Model) jest traktowany jako model odniesienia (wzorzec) dla większości rodzin protokołów komunikacyjnych. Podstawowym założeniem modelu jest podział systemów sieciowych na 7 warstw (ang. layers) współpracujących ze sobą w ścisłe określony sposób. Przed wysłaniem dane wraz z przekazywaniem do niższych warstw sieci zmieniają swój format, co nosi nazwę procesu kapsulkowania (enkapsulacji). Każda warstwa posiada własne jednostki przesyłanych danych (PDU, ang. protocol data unit).

Protokół komunikacyjny to zbiór ścisłych reguł i kroków postępowania, które są automatycznie wykonywane przez urządzenia komunikacyjne w celu nawiązania łączności i wymiany danych.

Pomimo, że model OSI jest omawiany na wszystkich zajęciach (oraz we wszystkich książkach) dotyczących sieci, to postaram się dogłębiego tutaj opisać na podstawie książki [1]. Jeśli model OSI jest Ci znany, to możesz pominąć ten podrozdział.

Założmy, że stoją przed nami dwa komputery, które chcemy połączyć tak, żeby mogły się ze sobą komunikować. Możemy w tym celu skorzystać z kabla, czy równie dobrze z fal radiowych. W zależności od wybranego sposobu przesyłania danych musimy ustalić w jaki sposób będzie to zrealizowane. Trzeba zdecydować jak będą odbierane i wysyłane fale radiowe, żeby przekazać bity i bajty. Oprócz tego należy zdefiniować złącza (interfejsy) oraz mechanizmy

rozwiązuje problemy wynikające z interferencji fal. Ta część modelu OSI to **warstwa pierwsza (fizyczna)**, gdzie przesyłanymi danymi są zera i jedynki.

Warstwa druga (łącza danych) rozwiązuje problemy pojawiające się, gdy sieć się rozrasta. W sieci składającej się z 2 komputerów sprawa jest prosta. Wszystko to, co wysyła pierwszy komputer przesyłamy do drugiego i odwrotnie. Jednakże w jaki sposób będziemy decydować, do kogo mają być przesłane dane, gdy w sieci jest więcej niż 2 komputery? Odpowiedzą na to pytanie jest adresacja, czyli przydzielenie każdemu komputerowi unikalnego adresu. Drugi problem to współdzielenie medium do przesyłania danych, czyli rozwiązanie problemu, gdy dwa komputery chcą nadawać dane w tym samym momencie. Druga warstwa zajmuje się pakowaniem danych w ramki (ang. frame) i wysyłaniem do warstwy fizycznej. Rozpoznaje błędy związane z gubieniem pakietów oraz uszkodzeniem ramek i zajmuje się ich naprawą. Ramka zawiera adres sieciowy (adres MAC) nadawcy i odbiorcy.

Kolejnym krokiem jest wysłanie danych do innej sieci. Każdy komputer zna adresy innych komputerów we własnej sieci, jednakże analogiczne rozwiązanie dla komunikacji między komputerami znajdującymi się w różnych sieciach nie jest możliwe, bo ciężko sobie wyobrazić, że komputer zna adres wszystkich innych komputerów na świecie. W tym momencie pojawia się **warstwa trzecia (sieciowa)**, która wprowadza pakiety (ang. packet), zawierające adres sieci oraz informacje o trasowaniu (ang. routing). Jeśli przyjmiemy, że naszą siecią jest nasze miasto, to adres sieciowy komputera (z warstwy drugiej) to ulica oraz numer budynku (z numerem mieszkania), zaś adres sieci (z warstwy trzeciej) to kod pocztowy miasta.

Oprócz możliwości przesyłania danych między różnymi sieciami istotne jest również zapewnienie jakości i niezawodności komunikacji, dzięki której mamy pewność, że nasze dane (pakiety) zostały pomyślnie dostarczone do odbiorcy. Druga sprawa, to możliwości wykorzystywanej medium. Nie chcemy przesyłać danych z taką prędkością, że medium nie jest ich w stanie obsłużyć. Dlatego dane dzielone są na mniejsze porcje, którymi łatwiej zarządzać. To jest zadanie **warstwy czwartej (transportowej)**, która operuje na segmentach danych i zapewnia niezawodność komunikacji między komunikującymi się ze sobą komputerami. Mechanizmy gwarantujące niezawodność komunikacji to retrasmisja zagubionych lub uszkodzonych pakietów oraz kontrola przepływu danych, której zadaniem jest dopilnowanie, aby pakiety były przesyłane z taką prędkością, że obie strony uczestniczące w komunikacji będą nadążały je przetwarzać.

Powyższe cztery warstwy to tzw. warstwy niższe, które zajmują się odnajdywaniem odpowiedniej drogi do celu, gdzie ma być przekazana konkretna informacja. Ważną cechą warstw dolnych jest całkowite ignorowanie sensu przesyłanych danych. Dla warstw dolnych nie istnieją aplikacje, tylko segmenty/pakiety/ramki danych.

Kolejne warstwy modelu OSI to warstwy wyższe (warstwy danych) i ich zadaniem jest współpraca z oprogramowaniem realizującym zadania zlecone przez użytkownika systemu komputerowego. One skoncentrowane są na danych.

Warstwa piąta (sesji) nie zmienia przesyłanych danych, a jedynie zarządza (otwiera, utrzymuje, zamyka) sesję. Otrzymuje od różnych aplikacji dane,

Warstwa OSI	Przykłady protokołów	PDU
Aplikacji	FTP, HTTP, ...	Dane
Prezentacji	MIME, NCP, ...	
Sesji	SCP, ZIP, ...	
Transportowa	TCP, UDP	Segment
Sieci	IP	Pakiet
Łącza danych	ARP, PPP, ...	Ramka
Fizyczna	DSSS(Wifi), Standardy USB, Bluetooth, ...	Bit

Tablica 1.1: Model OSI

które muszą zostać odpowiednio zsynchronizowane. Synchronizacja występuje między warstwami sesji systemu nadawcy i odbiorcy. Warstwa sesji „wie”, która aplikacja łączy się z którą, dzięki czemu może zapewnić właściwy kierunek przepływu danych – nadzoruje połączenie. Wznawia je po przerwaniu.

Zadaniem **warstwy szóstej (prezentacji)** jest przetworzenie danych od aplikacji do postaci kanonicznej (ang. canonical representation) zgodnej ze specyfikacją modelu OSI, dzięki czemu niższe warstwy zawsze otrzymują dane w tym samym formacie. Wynika to ze zróżnicowania systemów komputerowych, które mogą w różny sposób interpretować te same dane. Na przykład bity w bajcie danych w różnych procesorach są interpretowane w odwrotnej kolejności niż w innych (big i little endian). Innym przykładem jest wiadomość e-mail, która stworzona w programie MS Outlook mogłaby być niezrozumiała dla programu Thunderbird, dlatego w warstwie prezentacji jest przetłumaczona na taką wersję zrozumiałą dla każdego odbiorcy, czyli kod ASCII. Podsumowując, warstwa ta odpowiada za kodowanie i konwersję danych oraz za kompresję / dekompresję; szyfrowanie / deszyfrowanie.

Najwyższa, **siódma warstwa (aplikacji)** zajmuje się specyfikacją interfejsu, który wykorzystują aplikacje do przesyłania danych do sieci (poprzez kolejne warstwy modelu ISO/OSI). W przypadku sieci komputerowych aplikacje są zwykle procesami uruchomionymi na odległych hostach. Interfejs udostępniający programistom usługi dostarczane przez warstwę aplikacji opiera się na obiektach nazywanych gniazdami (ang. socket). W tej warstwie występują wszystkie protokoły, które pozwalają użytkownikowi na dostęp do przesyłanych danych. Na przykład protokół FTP pozwala na przesyłanie plików, SMTP pozwala na przesyłanie wiadomości e-mail, a HTTP pozwala na serfowanie po Internecie.

1.2 Model TCP/IP

Model OSI jest standardowym modelem komunikacji między systemami komputerowymi, jednakże podstawą struktury Internetu stał się model TCP/IP, który został stworzony w latach 70. XX wieku w DARPA (ang. Defense Advanced Research Projects Agency), aby pomóc w tworzeniu odpornych na atak sieci

OSI	TCP/IP	Protokoły
Aplikacji	Aplikacji	FTP, HTTP, SMTP, DNS, POP, IMAP, Telnet, SSH, DHCP, ...
Prezentacji		
Sesji		
Transportowa	Transportowa	TCP, UDP
Sieci	Internetu	IP
Łącza danych	Dostępu do sieci	ARP, STP, FDDI, ...
Fizyczna		

Tablica 1.2: Model TCP/IP

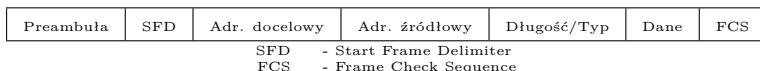
komputerowych.

TCP/IP jest zbiorem protokołów komunikacyjnych, które umożliwiają komunikację węzłom (tzw. hostom) w sieci, który podobnie jak model OSI zorganizowany jest w strukturze warstwowej, gdzie każda warstwa realizuje odrębne zadanie. Jednak liczba warstw jest mniejsza i bardziej odzwierciedla prawdziwą strukturę Internetu. Tabela 1.2 przedstawia stos TCP/IP.

Datagram to blok danych pakietowych przesyłany przez sieć komunikacyjną między komputerami lub abonentami sieci, zawierający wszelkie niezbędne informacje do przesłania danych z hosta źródłowego do hosta docelowego, bez konieczności wcześniejszej wymiany informacji przez te hosty.

Za Wikipedią opiszmy wszystkie warstwy modelu TCP/IP:

- Warstwa procesowa czy warstwa aplikacji (ang. process layer) to najwyższy poziom, w którym pracują użyteczne dla człowieka aplikacje takie jak np. serwer WWW czy przeglądarka internetowa. Obejmuje ona zestaw gotowych protokołów, które aplikacje wykorzystują do przesyłania różnego typu informacji w sieci. Wykorzystywane protokoły to m.in.: HTTP, Telnet, FTP, TFTCP, SNMP, DNS, SMTP, X Window.
- Warstwa transportowa (ang. host-to-host layer) gwarantuje pewność przesyłania danych oraz kieruje właściwe informacje do odpowiednich aplikacji. Opiera się to na wykorzystaniu portów określonych dla każdego połączenia. W jednym komputerze może istnieć wiele aplikacji wymieniających dane z tym samym komputerem w sieci i nie nastąpi wymieszanie się przesyłanych przez nie danych. To właśnie ta warstwa nawiązuje i zrywa połączenia między komputerami oraz zapewnia pewność transmisji.
- Warstwa Internetu lub warstwa protokołu internetowego (ang. internet protocol layer) to sedno działania Internetu. W tej warstwie przetwarzane są datagramy posiadające adresy IP. Ustalana jest odpowiednia droga do docelowego komputera w sieci. Niektóre urządzenia sieciowe posiadają tę warstwę jako najwyższą. Są to routery, które zajmują się kierowaniem



Rysunek 1.1: Ramka (Ethernet)

ruchu w Internecie, bo znają topologię sieci. Proces odnajdywania przez routery właściwej drogi określa się jako trasowanie.

- Warstwa dostępu do sieci lub warstwa fizyczna (ang. network access layer) jest najniższą warstwą i to ona zajmuje się przekazywaniem danych przez fizyczne połączenia między urządzeniami sieciowymi. Najczęściej są to karty sieciowe lub modemy. Dodatkowo warstwa ta jest czasami wyposażona w protokoły do dynamicznego określania adresów IP.

Łatwo widać, że model TCP/IP był wzorowany na modelu OSI, a dokładniej jest jego uproszczoną wersją.

Protokoły wspominane w przypadku użycia (TCP, ARP, DNS) zostaną omówione szczegółowo później.

Prześledźmy teraz konkretny przypadek użycia modelu TCP/IP. Założymy, że Joe chce zrobić zakupy przez Internet. Joe otwiera przeglądarkę i wpisując adres sklepu tworzy zapytanie. Jego komputer przyjmuje od niego zapytanie i stwierza, że nie może go obsłużyć wewnętrznie (bez komunikacji z innymi komputerami). Komputer musi znaleźć serwer, który odpowie na żądanie i na początku ustala, że protokół obsługujący to żądanie to HTTP (przecież pochodzi z przeglądarki), po czym zaczyna tworzyć sesję przesyłając pakiety do serwera i spowrotem według poniższego opisu.

Komputer Joe znajduje się w sieci Ethernet, przez co korzysta ze stosu TCP/IP, czyli komunikując się z innymi komputerami porządkuje bity według ustalonych formatów danych - ramek. Te ramki (widoczne na rysunku 1.1) budowane są przez kolejne warstwy począwszy od warstwy najwyższej (applikacji), a na najniższej skończywszy (dostępu do sieci). Warstwa aplikacji przekaże zapytanie HTTP (dane) do warstwy niższej. Teraz komputer sprawdzi zapytanie HTTP i decyduje, że w dalszej kolejności będzie korzystał z sesji utrzymywającej połączenie (ang. connection-oriented), ponieważ gwarantuje ona niezawodność komunikacji, przez co Joe otrzyma dokładnie to, czego oczekuje i nic nie zostanie pominięte.

Na poziomie warstwy transportowej komputer wybiera protokół TCP (ang. Transmission Control Protocol), który wymieni z serwerem serię pakietów w celu nawiązania połączenia. Ta wymiana pakietów nazywa się 3-krokowym handshake'iem (ang. 3-step handshake), ponieważ polega na wymianie 3 pakietów synchronizujących i ustalających połączenie. Pierwszy z pakietów (tzw. SYN) jest przekazywany do warstwy niższej.

Warstwa Internetu musi ustalić adres IP serwera, z którym chce się połączyć. Joe podał w przeglądarce domenę skepu, jednakże nie jest ona adresem serwera, a jedynie jego aliasem. W związku z tym, komputer Joe musi skorzystać z innego protokołu (DNS) i zapytać jaki jest adres domeny, którą podał Joe. Kiedy komputer otrzyma odpowiedź z adresem IP, warstwa Internetu przygotowuje pakiet, który zawiera dane, nagłówek synchronizujący i pozostałe informację dotyczące protokołu IP, po czym przekazuje go do warstwy niższej.

Na poziomie warstwy dostępu do sieci komputer Joe musi ustalić adres lokalny (fizyczny, MAC) komputera, do którego powinien przesłać ramkę w obrębie własnej sieci. Komputer zna już adres IP serwera, jednakże na poziomie tej warstwy wymagany jest adres lokalny. Komputer nie skomunikuje się on bezpośrednio z serwerem odpowiadającym na zapytanie HTTP, lecz raczej stworzona przez niego ramka zostanie przekazana do jego bramy domyślnej, która przesła pakiet dalej. W celu uzyskania adresu lokalnego, komputer Joe skorzysta z kolejnego protokołu (ARP), dzięki któremu uzyska adres fizyczny (w tym przypadku adres fizyczny bramy domyślnej).

W przypadku komunikacji z innymi sieciami pakiety przesyłane są przez bramę domyślną i komputery w obrębie sieci lokalnej znają jej adres fizyczny. W związku z tym protokół ARP nie zostanie wykorzystany. Jednakże jest on jak najbardziej potrzebny, gdy chcemy się skomunikować z innym komputerem z naszej sieci lokalnej.

Po uzyskaniu adresu lokalnego ramka z adresami lokalnymi i danymi pochodzącyymi z warstw wyższych może opuścić komputer.

Proces odpytywania o adres lokalny jest powtarzany na każdym etapie przekazywania pakietu, aż dotrze on do maszyny docelowej (serwera). W każdym kroku z ramki wypakowywane są dane z warstw wyższych, komputer przekazujący odpytuje o kolejny adres lokalny korzystając z ARP, tworzy nową ramkę z danymi i przesyła dalej.

Na końcu pakiet dociera do komputera docelowego, czyli serwera, gdzie przekazywany jest po kolei do wyższych warstw tak, żeby do aplikacji serwera WWW dotarło samo zapytanie HTTP. Po przetworzeniu zapytania, serwer w analogicznym procesie odpowiada, wysyłając pakiet do komputera Joe.

2

Protokoły komunikacyjne

W tym rozdziale omówimy podstawowe protokoły z warstw niższych oraz popularne protokoły z warstwy aplikacji.

2.1 Podgląd pakietów

Na początku poznamy sposób podsłuchiwanego (ang. sniffing) i przeglądania pakietów. W części zadań dotyczących protokołów przygotowany będzie plik z pakietami, w pozostałych natomiast trzeba będzie je samemu przechwycić.

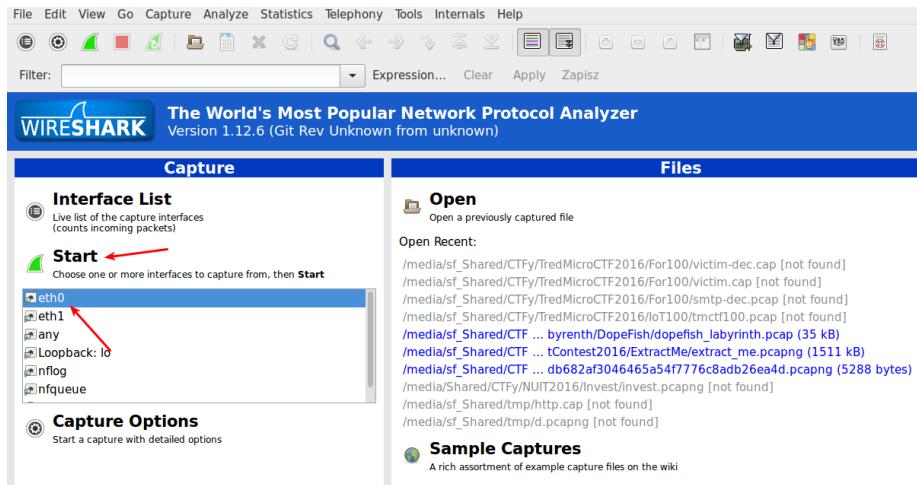
Do podsłuchiwanego pakietów można wykorzystać narzędzie *tcpdump* (konsolowe) lub *Wireshark* (GUI). Generują one plik w formacie pcap (lub pcapng), który zawiera podsłuchane pakiety. W dalszej części skryptu będę wykorzystywał program *Wireshark* ze względu na jego wygodę w użyciu, w szczególności do przeglądania pakietów (wersja z interfejsem użytkownika) oraz na bogatą bibliotekę dekodowania pakietów.

Pełny podręcznik użytkownika programu *Wireshark* można znaleźć na stronie https://www.wireshark.org/docs/wsug_html_chunked/.

2.1.1 Podsłuchiwanie pakietów

Pierwszym krokiem jest uruchomienie programu *Wireshark*. Na głównym ekranie (Rysunek 2.1) należy wybrać interfejs, na którym chcemy podsłuchiwać pakiety i uruchomić proces przyciskiem Start.

Po uruchomieniu podsłuchiwanego pakiety są dekodowane i wyświetlane na ekranie w czasie rzeczywistym (Rysunek 2.2). Ekran podzielony jest na trzy części. W górnej części umieszczone są przechwycone pakiety uporządkowane zgodnie z czasem ich nadania. Poniżej znajduje się panel, w którym rozkodowana jest struktura pakietu, która jest zależna od protokołu, którego pakiet



Rysunek 2.1: Wireshark. Okno główne.

dotyczy. *Wireshark* sam rozpoznaje protokół i na tej podstawie wykorzystuje i uzupełnia właściwą strukturę.

Na Rysunku 2.2 w górnym panelu zaznaczono pakiet przesłany protokołem HTTP. *Wireshark* rozpoznał protokół i dlatego w kolumnie *Protocol* umieścił jego nazwę. Dzięki znajomości protokołu *Wireshark* wie, do jakiej struktury przypisać bajty przesyłane w pakiecie, co widać w panelu środkowym. *Wireshark* umieszcza w niej listę rozpoznanych protokołów i formatów, a wśród nich znajduje się protokół HTTP, którym przesłany został plik w formacie JPEG.

Oczywiście protokół HTTP pochodzi z warstwy aplikacji, a na warstwach niższych wykorzystywane są inne protokoły, co widać w środkowym panelu na Rysunku 2.2 w postaci wpisów rozpoczynających się od nazwy protokołu (np. Internet protocol). Protokoły te zostaną omówione w dalszej części.

Ostatni panel to podgląd w formacie HEX (szestastkowym) oraz formacie ASCII treści całego pakietu. Zaznaczenie jednej z sekcji w panelu środkowym podświetla bajty w panelu dolnym, które składają się na tę sekcję.

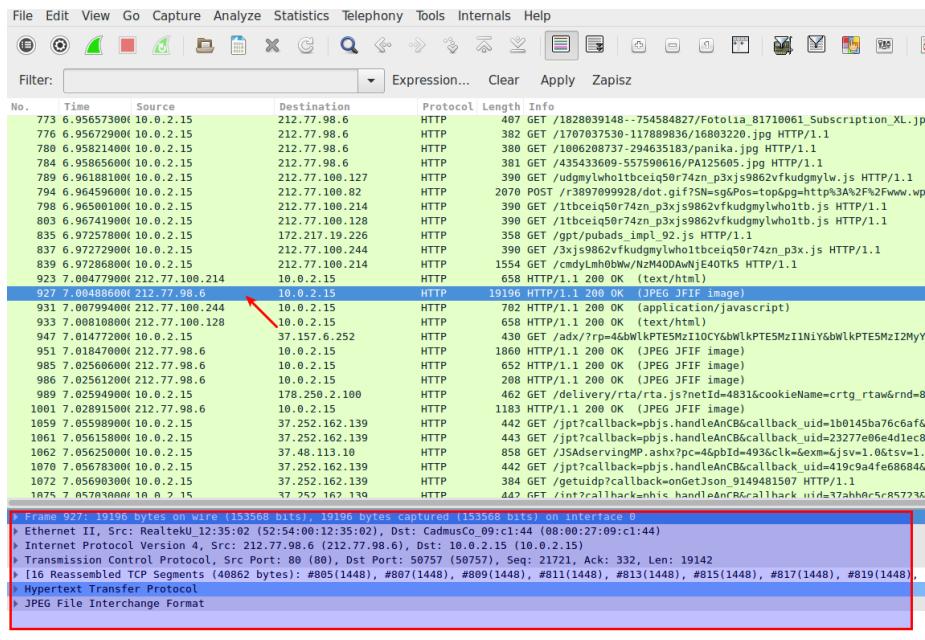
2.1.2 Filtry

W niektórych przypadkach do przeanalizowania będzie przygotowany plik z podsłuchanymi pakietami. Wtedy należy ten plik wczytać w *Wiresharku*, korzystając z menu *Plik > Otwórz* (*File > Open*).

Ręczny przegląd wszystkich pakietów, tak jak opisano w poprzednim podrozdziale, jest niemożliwy w przypadku plików zawierających setki tysięcy pakietów. Zwykle jest tak, że plik (tzw. „źrzut”) zawiera wszystkie podsłuchane pakiety, a nas interesuje tylko niewielki ich zbiór (np. analizowany przez nas protokół). W takiej sytuacji pomocne są filtry, które wyświetlają tylko wybrany zbiór pakietów. Przykład filtru wyświetlającego tylko zapytania i odpowiedzi

2.1. PODGLĄD PAKIETÓW

9



Rysunek 2.2: Wireshark. Podgląd pakietów.

No.	Time	Source	Destination	Protocol	Length	Info
681	4.837398900	213.180.141.128	10.0.2.15	HTTP	2377	HTTP/1.1 200 OK (JPEG JFIF image)
685	5.032781000	10.0.2.15	213.180.141.128	HTTP	421	GET /paas-static/template-engine/c52c1l
705	5.069583000	213.180.141.128	10.0.2.15	HTTP	1119	HTTP/1.1 200 OK (image/x-icon)
731	5.517448000	10.0.2.15	213.180.141.148	HTTP	361	GET /s.csr/v1/build/dlApi/dl_sg.utils.r
736	5.526587000	10.0.2.15	213.180.139.227	HTTP	536	GET /fpdata.js?href=www.onet.pl HTTP/1
756	5.542689000	213.180.139.227	10.0.2.15	HTTP	710	HTTP/1.1 200 OK (application/x-javasc
792	5.564870000	213.180.141.148	10.0.2.15	HTTP	854	HTTP/1.1 200 OK (application/javascript)
803	5.691086000	10.0.2.15	216.58.201.226	HTTP	399	GET /tag/jst/gpt.js HTTP/1.1
807	5.701782000	10.0.2.15	172.217.20.238	OCSP	500	Request
813	5.702873000	10.0.2.15	213.180.141.150	HTTP	380	GET /_s.csr-006/csr.js?site=GLOWNA&are
820	5.717857000	216.58.201.226	10.0.2.15	HTTP	639	HTTP/1.1 200 OK (text/javascript)
822	5.735453000	172.217.20.238	10.0.2.15	OCSP	800	Response

Rysunek 2.3: Wireshark. Filtr HTTP.

wysłane po protokole HTTP przedstawiono na Rysunku 2.3.

Filtryle umieszczane są w polu tekstowym nad głównym panelem (zaznaczone na Rysunku 2.3). Ich składnia jest rozbudowana i spis wszystkich możliwych filtrów można znaleźć na stronie <https://www.wireshark.org/docs/dref/>.

Do popularnych składni filtrów należą:

- **Wybór pakietów z konkretnego protokołu.** Filtr `http` wyświetli tylko te pakiety, które przez *Wiresharka* zostały przypisane do protokołu HTTP.
- **Porównanie wartości pola protokołu.** Filtr `tcp.port eq 25` lub `tcp.port == 25` wyświetli pakiety, które zostały wysłane z lub na port 25 protokołem TCP. Pole `port` nie jest dostępne we wszystkich protokołach, dlatego należy wybrać protokół, w którym to pole istnieje. Natomiast filtr `ip.src==192.168.0.0/16` wyświetli pakiety, które zostały wysłane z adresu IP pochodzącego z sieci 192.168.0.0/16.
- **Porównanie wartości z wyrażeniem regularnym.** Filtr `http.request.uri matches "php$"` wyświetli te zapytania HTTP, w których adres URL kończy się znakami php.
- **Łączenie wyrażeń spójnikami logicznymi.** Filtr `tcp.port == 80 || tcp.port == 25` wyświetli pakiety wysyłane z lub na porty 25 lub 80, natomiast `tcp.flags.syn && tcp.flags.ack` wyświetli te pakiety, które mają ustawione flagi SYN i ACK.
- **„Surowe” porównanie bajtów.** Nie jest to bardzo popularny filtr, jednakże wpominam o nim ponieważ przyda się na pewno do analizy własnych protokołów, które nie są dekodowane przez *Wiresharka*. Na przykład filtr `udp[8:3]==81:60:03` wyświetli te pakiety, w których dane w protokole UDP rozpoczynają się od 3 wymienionych bajtów, wpisywanych w trybie szesnastkowym.

No.	Time	Source	Destination	Protocol	Length	Info
39	4.580963000 CadmusCo_	Broadcast		ARP	42	Who has 10.0.2.2? Tell 10.0.2.15
40	4.581059000 RealtekU_	CadmusCo_09:		ARP	60	10.0.2.2 is at 52:54: RealtekU_12: [REDACTED]
203	5.012152000 CadmusCo_	RealtekU_12:		ARP	42	Who has 10.0.2.3? Tell 10.0.2.15
204	5.013586000 RealtekU_	CadmusCo_09:		ARP	60	10.0.2.3 is at 52:54

Rysunek 2.4: Wireshark. ARP - odfiltrowane pakiety.

2.2 ARP

Pierwszym omówionym protokołem będzie protokół ARP (ang. Address Resolution Protocol), pochodzący z Warstwy Łącza Danych w modelu OSI (Warstwy Dostępu do Sieci w modelu TCP/IP). ARP umożliwia mapowanie logicznych adresów Warstwy Sieciowej (warstwa 3) na fizyczne adresy Warstwy Łącza Danych.

Na Rysunku 2.4 przedstawiono pakiety przesłane w ramach protokołu ARP. Pierwszy z nich to zapytanie o adres fizyczny komputera, który ma przypisany adres IP 10.0.2.2. Zapytanie wysłane jest przez komputer z kartą sieciową firmy Cadmus do wszystkich (Broadcast), ponieważ maszyna nie wie, do kogo ma wysłać zapytanie.

Adres fizyczny MAC karty sieciowej składa się z 6 bajtów (48 bitów). Pierwsze 24 bity oznaczają producenta karty sieciowej, pozostałe 24 bity są unikatowym identyfikatorem danego egzemplarza karty.

Drugi pakiet to odpowiedź od komputera z kartą firmy Realtek, skierowaną już bezpośrednio do pytającego, czyli wyżej wspomnianego komputera (Cadmus). W odpowiedzi umieszczony jest adres fizyczny komputera o zadanym adresie IP. Następne pakiety to zapytania i odpowiedzi o kolejne adresy IP. W nich fizyczny adres docelowy na poziomie protokołu Ethernet z formatem ramki w wersji II jest już ustalony (nie jest to Broadcast), ponieważ został zapamiętany po wcześniejszym wysłanym zapytaniu.

Protokół ARP definiuje zapisywanie (tzw. cacheowanie) otrzymanych adresów fizycznych wraz z przypisanym do nich adresem IP w tablicy lokalnej w celu zmniejszenia liczby emitowanych zapytań.

2.2.1 Format ARP

Format pakietu ARP przedstawiono w Tabeli 2.1.

Poszczególne pola mają następującą definicję:

- **Typ warstwy fizycznej (HTYPE)** - typ protokołu warstwy fizycznej.
Możliwe wartości dla pola HTYPE to:

– 1 Ethernet

Bajty	0	1
0	Typ warstwy fizycznej (HTYPE)	
2	Typ protokołu wyższej warstwy (PTYPE)	
4	Długość adresu sprzętowego (HLEN)	Długość protokołu wyższej warstwy (PLEN)
6	Operacja (OPER)	
8	Adres sprzętowy źródła (SHA)	
14/?	Adres protokołu wyższej warstwy źródła (SPA)	
18/?	Adres sprzętowy celu (THA)	
24/?	Adres protokołu wyższej warstwy celu (TPA)	

Tablica 2.1: Format pakietu ARP.

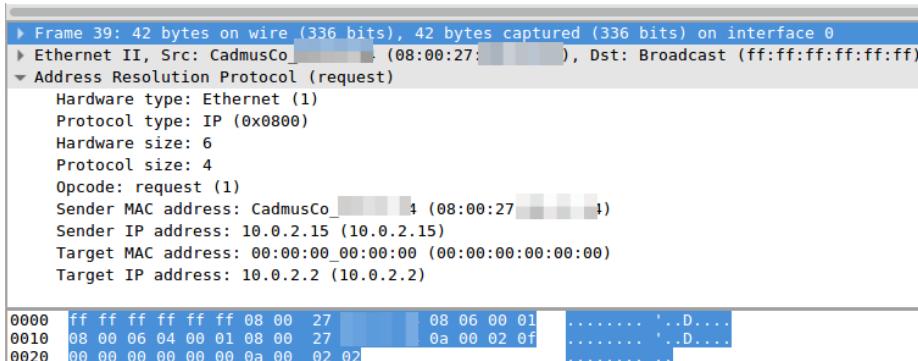
- 6 IEEE 802.3
- 15 Frame Relay
- 16 ATM
- 17 HDLC
- 18 Fibre Channel
- 19 ATM
- 20 Serial Line
- 30 ATM
- 31 IPsec

- **Typ protokołu wyższej warstwy (PTYPE)** - dla protokołu IPv4 jest to 0x0800. Dopuszczalne wartości to wartości z tabeli EtherType.
- **Długość adresu sprzętowego (HLEN)** - długość adresu sprzętowego (MAC) podana w bajtach.
- **Długość protokołu wyższej warstwy (PLEN)** - długość adresu protokołu, np. IP (4 bajty).
- **Operacja (OPER)** - kod operacji ARP. Poniżej cztery najważniejsze wartości:
 - 1 Zapytanie
 - 2 Odpowiedź
 - 3 Zapytanie odwrotne
 - 4 Odpowiedź odwrotna
- **Adres sprzętowy źródła (SHA)** - sprzętowy adres (MAC) nadawcy.
- **Adres protokołu wyższej warstwy źródła (SPA)** - adres protokołu warstwy wyższej nadawcy, np. adres IP.
- **Adres sprzętowy przeznaczenia (THA)** - sprzętowy adres (MAC) odbiorcy.

- **Adres protokołu wyższej warstwy przeznaczenia (TPA)** - adres protokołu warstwy wyższej odbiorcy, np. adres IP.

Protokół ARP nie ogranicza się tylko do konwersji adresów IP na adres MAC stosowany w sieciach Ethernet, lecz jest także wykorzystywany do odpytywania o adresy fizyczne stosowane w innych technologiach, o czym świadczy pole PTYPE. Wszystkie możliwe wartości pola PTYPE można znaleźć na stronie <http://www.iana.org/assignments/ieee-802-numbers/ieee-802-numbers.xhtml>.

Na Rysunku 2.5 znajduje się przykład pakietu dla protokołu ARP. Widać na nim, że pole HTYPE ma wartość 1, rozpoznaną przez *Wireshark* jako typ Ethernet. Typ protokołu warstwy wyższej to 0x0800, czyli protokół IP. Kolejne dwa pola to odpowiednio długość adresu fizycznego (6 bajtów) oraz długość adresu IP (4 bajty). Następne pole to **Opcode** o wartości 1, czyli jest to zapytanie ARP. Kolejne 4 pola to adres fizyczny i adres z wyższej warstwy (w tym przypadku IP) nadawcy i odbiorcy. Długość tych pól jest zależna od wartości pól wspomnianych wyżej, stąd w Tabeli 2.1 umieściłem w pierwszej kolumnie znaki zapytania (wartości liczbowe odpowiadają sytuacji, gdy pytamy o adresy MAC na podstawie adresów IP). W przykładzie na Rysunku 2.5 wartość pola Target MAC address jest zerem (a właściwie sześcioma zerowymi bajtami), co ma sens, bo przecież pytamy o adres MAC komputera, który ma adres IP 10.0.2.2.



Rysunek 2.5: Wireshark. ARP - przykład pakietu.

2.3 IP

Protokół IP (ang. Internet Protocol) to powszechnie wykorzystywany w sieci Internet protokół warstwy trzeciej modelu OSI, czyli Warstwy Sieci (Warstwy Internetowej w modelu TCI/IP). Jak wspomniałem w rozdziale 1.1, Warstwa

Sieci umożliwia komunikację między różnymi sieciami (jak również w sieci lokalnej) poprzez nadanie każdemu komputerowemu adresu logicznego oraz trasowanie, czyli takie przesyłanie pakietów między komputerami, żeby dotarły do komputera docelowego.

Adres logiczny z Warstwy Sieci znany jako adres IP to właśnie adres logiczny przydzielany przez protokół IP, który określa jego format. W przedstawionych przykładach będzie wykorzystany protokół IP w wersji 4, gdzie adres logiczny to liczba całkowita z przedziału $0 - 2^{32}$, zwykle podzielona na 4 bajty.

2.3.1 Klasy IP

W pierwotnej specyfikacji protokołu wprowadzono podział adresów na klasy, przydzielając pule adresów według wartości pierwszego bajtu do klas A, B, C, D lub E. Przynależność do danej klasy określała rozmiar maski sieci [3].

RFC (ang. Request for Comments – dosłownie: prośba o komentarze) – zbiór technicznych oraz organizacyjnych dokumentów mających formę memorandum związań z Internetem oraz sieciami komputerowymi. Każdy z nich ma przypisany unikatowy numer identyfikacyjny, zwykle używany przy wszelkich odniesieniach.

Dokumenty nie mają mocy oficjalnej, jednak niektóre z nich zostały później przekształcone w oficjalne standardy sieciowe, np. opis większości popularnych protokołów sieciowych został pierwotnie opisany właśnie w RFC.

W związku z wprowadzeniem od roku 1993 założeń RFC 1518 i RFC 1519 (późniejszy RFC 4632), dotyczących wprowadzenie routingu bez klas (CIDR, ang. Classless Inter-Domain Routing), podział na klasy A, B i C utracił znaczenie przy ustalaniu rozmiaru maski sieci i routing w sieciach IP opiera się obecnie na podawanych w konfiguracji maskach sieci bez uwzględniania klas adresowych. Adresy klasy D w dalszym ciągu jest przypisane do usług multikastowych.

Oktet to jednostka rozmiaru danych, która wynosi 8 bitów, czyli tyle ile bajt.

- **Klasa A** - Do identyfikacji sieci wykorzystany jest wyłącznie pierwszy oktet, pozostałe trzy stanowią adres hosta. Najstarszy bit pierwszego bajtu adresu jest zawsze równy zeru, ponadto liczby 0 i 127 są zarezerwowane, dlatego ostatecznie dostępnych jest 126 adresów sieci tej klasy. Klasa ta została przeznaczona dla wyjątkowo dużych sieci, ponieważ trzy ostatnie oktety bajtów adresu dają ponad 16 milionów numerów hostów. Szczególnym przypadkiem jest sieć, w której pierwszy bajt to 127, która

jest wykorzystywana przez komputer do komunikacji z samym sobą (pętla zwrotna, ang. loopback).

- **Klasa B** - Pierwsze dwa oktety opisują adres sieci tej klasy, pozostałe określają adres hosta. Najstarsze dwa bity pierwszego bajtu adresu to 10, dlatego może on zawierać 63 kombinacji (od 128 do 191), drugi może być dowolny dając tym samym do dyspozycji ponad 16 tysięcy adresów sieci. W każdej z sieci można przypisać podobną liczbę hostów (ponad 65 tysięcy), z tego powodu klasa ta została przeznaczona dla potrzeb sieci średnich i dużych.
- **Klasa C** - Trzy pierwsze bajty opisują adres sieci, przy czym pierwszy z nich zawsze zaczyna się kombinacją dwójkową 110. Pierwszy bajt pozwala na przypisanie 31 kombinacji (od 192 do 223), kolejne dwa mogą być przypisane dowolnie, dając ostatecznie ponad 2 miliony adresów sieci. Ostatni oktet przeznaczony jest do określenia adresu hosta w sieci. Maksymalnie może być ich 254 (bez 0 oraz 255), dlatego ta przestrzeń adresowa została przeznaczona dla małych sieci.
- **Klasa D** - Pierwsze cztery bity adresu tej klasy wynoszą 1110, stąd dostępne jest 16 kombinacji (od 224 do 239) dla pierwszego oktetu. Ta przestrzeń adresowa została utworzona w celu umożliwienia rozsyłania grupowego przy użyciu adresów IP. Adres rozsyłania grupowego jest unikatowym adresem sieciowym, który kieruje pakiety o tym adresie docelowym do zdefiniowanej wcześniej grupy adresów IP. Dzięki temu pojedynczy komputer może przesyłać jeden strumień danych równocześnie do wielu odbiorców (multicast).
- **Klasa E** - Adresy tej klasy zostały zarezerwowane przez Internet Engineering Task Force (IETF) do potrzeb badawczych i nie są dostępne do publicznego użytku. Pierwsze cztery bity każdego adresu tej klasy mają zawsze wartość 1, dlatego istnieje tylko 15 możliwości (od 240 do 255) przypisania pierwszego bajtu.

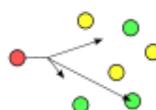
2.3.2 Trasowanie

Trasowanie (ang. routing), czyli wyznaczanie trasy i wysłanie nią pakietu danych w sieci komputerowej to druga z dwóch głównych funkcji protokołu IP, oprócz adresowania. Urządzenie węzłowe, w którym kształtowany jest ruch sieciowy, nazywane jest routерem, a jego rolę może pełnić np. komputer stacjonarny czy oddzielne dedykowane urządzenie.

Typy trasowania

Anycast - dane wysypane są do topologicznie najbliższego (czyli teoretycznie najlepszego) odbiorcy. Komunikacja następuje od jednego nadawcy do (po-

anycast



tencjalnie) wielu odbiorców, przy czym jednocześnie dane są odbierane przez jednego z nich.

Broadcast - rozgłoszeniowy tryb transmisji danych polegający na wysyłaniu przez jeden port (kanał informacyjny) pakietów, które powinny być odebrane przez wszystkie pozostałe porty przyłączone do danej sieci (domeny rozgłoszeniowej).

broadcast

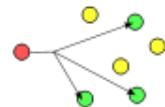


Multicast - sposób dystrybucji informacji, dla którego liczba odbiorców może być dowolna. Odbiorcy są widziani dla nadawcy jako pojedynczy grupowy odbiorca (host group) dostępny pod jednym adresem dla danej grupy multikastowej. Multicast różni się od unicastu zasadą działania i wynikającą stąd efektywnością. Największe oszczędności łączą multicast oferując tam gdzie rozmiary komunikatów są największe, czyli na przykład w transmisjach telekonferencyjnych, przesyłaniu sygnału radiowego i telewizyjnego.

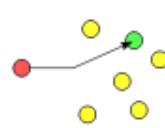
Unicast - dokładnie jeden punkt wysyła pakiety do dokładnie jednego punktu. Istnieje tylko jeden nadawca i tylko jeden odbiorca. Wszystkie karty Ethernet posiadają zaimplementowany ten rodzaj transmisji. Oparte na nim są podstawowe protokoły takie jak TCP, HTTP, SMTP, FTP i telnet i częściowo ARP, który pierwsze żądanie wysyła zawsze korzystając z transmisji broadcast.

Geocast - sposób dostarczania informacji w sieci komputerowej do grupy użytkowników, określonej przez położenie geograficzne. Jest to szczególny przypadek multycastu, wykorzystywany przez niektóre protokoły trasowania w sieciach typu Ad-Hoc. Położenie geograficzne określone jest jednym z trzech sposobów: punktem o znanych współrzędnych, kołem o znanym środku i promieniu albo wielokątem.

multicast



unicast



geocast



2.3.3 Format IP

Format pakietu IP przedstawiono w Tabeli 2.2.

Poszczególne pola mają następującą definicję:

- **Wersja (4 bity)** - (ang. Version) pole opisujące wersję protokołu, jednoznacznie definiujące format nagłówka. W naszym przypadku będzie to wartość 4 dla protokołu IPv4.

Bity	0-3	4-7	8-15	16-18	19-31		
0	Wersja	Długość nagłówka	Typ obsługi	Całkowita długość			
32	Numer identyfikacyjny		Flagi	Przesunięcie			
64	Czas życia		Protokół warstwy wyższej	Suma kontrolna nagłówka			
96	Adres źródłowy IP						
128	Adres docelowy IP						
160	Opcje IP			Wypełnienie			
192	Dane						

Tablica 2.2: Format pakietu IP.

- **Długość nagłówka (4 bity)** - (ang. Internet Header Length) długość nagłówka IP wyrażona w 32-bitowych słowach; minimalny, poprawny nagłówek ma długość co najmniej 5.
- **Typ usługi (8 bitów)** - (ang. Type of Services) pole wskazujące jaka jest pożądana wartość QoS dla danych przesyłanych w pakiecie. Na podstawie tego pola, routery ustawiają odpowiednie wartości transmisji. Pole jest definiowane na 5 różnych sposobów. Obecnie pole jest definiowane jako 5 podpól: 3 pierwsze bity zawierają wartości od 0 do 7 i używane są do oznaczania ważności datagramu. Domyślną wartością jest 0 (im wartość wyższa tym datagram ważniejszy). Bity 3 4 5 opisują co następuje: D: wymagane jest małe opóźnienie, T: wymagana jest duża przepustowość, R: wymagana jest duża niezawodność. Ostatnie dwa bity zarezerwowane są na pole ECN, które umożliwia zgłoszenie (między komunikującymi się komputerami) przeciążenia sieci bez pomijania pakietów. ECN to opcjonalna funkcja, która może być używany między dwoma punktami końcowymi ECN.
- **Całkowita długość pakietu (16 bitów)** - (ang. Total Length) długość całego datagramu IP (nagłówek oraz dane); maksymalna długość datagramu wynosi $2^{16} - 1 = 65535$. Minimalna wielkość datagramu jaką musi obsłużyć każdy host wynosi 576 bajtów, dłuższe pakiety mogą być dzielone na mniejsze (fragmentacja).
- **Numer identyfikacyjny (16 bitów)** - (ang. Identification) numer identyfikacyjny, wykorzystywany podczas fragmentacji do określenia przynależności pofragmentowanych datagramów do pakietów IP.
- **Flagi (3 bity)** - (ang. Flag) flagi wykorzystywane podczas fragmentacji datagramów. Zawierają dwa używane pola: DF (bit 2), które wskazuje, czy pakiet może być fragmentowany oraz MF (bit 3), które wskazuje, czy za danym datagramem znajdują się kolejne fragmenty. Bit 1 jest zarezerwowany.

- **Przesunięcie (13 bitów)** - (ang. Fragment Offset) w przypadku fragmentu większego datagramu pole to określa miejsce danych w oryginalnym datagramie; wyrażone w jednostkach ośmiooktetowych.
- **Czas życia (8 bitów)** - (ang. Time to live) czas życia datagramu. Zgodnie ze standardem liczba przeskoków przez jaką datagram znajduje się w obiegu. Jest zmniejszana za każdym razem, gdy datagram jest przetwarzany w routerze - jeżeli czas przetwarzania jest równy 0, datagram jest usuwany z sieci (nie przekazywany dalej) o czym nadawca usuniętego pakietu jest informowany zwrotnie z wykorzystaniem protokołu ICMP. Istnienie tej wartości jest konieczne, zapobiega krążeniu pakietów w sieci.
- **Protokół (8 bitów)** - (ang. Protocol) informacja o protokole, który jest przenoszony w polu danych datagramu IP. Dostępne wartości: https://en.wikipedia.org/wiki/List_of_IP_protocol_numbers. Protokoły TCP i UDP mają wartości odpowiednio 0x06 i 0x11. Protokół nie musi być protokołem warstwy wyższej, może być protokołem tej samej Warstwy Sieciowej, np. ICMP.
- **Suma kontrolna nagłówka (16 bitów)** - (ang. Header Checksum) suma kontrolna nagłówka pakietu, pozwalająca stwierdzić czy został on poprawnie przesłany, sprawdzana i aktualizowana przy każdym przetwarzaniu nagłówka.
- **Adres źródłowy (32 bity) i adres docelowy (32 bity)** - (ang. Source/Destination IP Address) pola adresów nadawcy i odbiorcy datagramu IP.
- **Opcje (32 bity)** - (ang. Options) niewymagane pole opcji, opisujące dodatkowe zachowanie pakietów IP. Wykorzystywane np. do debugowania czy pomiarów. Lista opcji dostępna pod adresem <http://www.networksorcery.com/enp/protocol/ip.htm#Options>.
- **Wypełnienie - (ang. Padding)** - opcjonalne pole wypełniające nagłówek tak, aby jego wielkość była wielokrotnością 32B, wypełnione zerami.

Sprójrzymy na podgląd pakietu IP podsłuchanego w Wiresharku, który umieściłem na Rysunku 2.6.

Widać na nim pakiet przesłany protokołem ICMP (tzw. ping) z Warstwy Sieciowej, który wykorzystuje protokół IP (również z Warstwy Sieciowej). W dolnej części są informacje o pakiecie IP, rozkodowane przez Wiresharka. Widać na rysunku, że wykorzystywany jest protokół IPv4, nagłówek ma 20 bajtów, a mechanizmy z pola Typ usługi nie są wykorzystywane (wszystkie są ustawione na zero). Cały pakiet IP (nagłówek IP oraz dane z warstw wyższych, bez danych z warstw niższych) zajmuje 84 bajty. Numer identyfikacyjny pakietu to 0x3f4a, jednakże flagi świadczą o tym, że pakiet nie został fragmentowany (wartość 0x02, czyli tylko 2 bit ustawiony). Czas życia protokołu to 64 tzw. hopy, a

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	CadmusCo	Broadcast	ARP	42	Who has 10.0.2.2? 1
2	0.000218000	RealtekU	CadmusCo	ARP	60	10.0.2.2 is at 52:54:b6:4d:00:00
3	0.000225000	10.0.2.15	8.8.8.8	ICMP	98	Echo (ping) request
4	0.032350000	8.8.8.8	10.0.2.15	ICMP	98	Echo (ping) reply
5	1.002214000	10.0.2.15	8.8.8.8	ICMP	98	Echo (ping) request
6	1.034067000	8.8.8.8	10.0.2.15	ICMP	98	Echo (ping) reply
7	2.003872000	10.0.2.15	8.8.8.8	ICMP	98	Echo (ping) request
8	2.037353000	8.8.8.8	10.0.2.15	ICMP	98	Echo (ping) reply

▼ Internet Protocol Version 4, Src: 10.0.2.15 (10.0.2.15), Dst: 8.8.8.8 (8.8.8.8)

- Version: 4
- Header Length: 20 bytes
- Differentiated Services Field: 0x00 (DSCP 0x00: Default; ECN: 0x00: Not-ECT (Not ECN-Capable Transport))
- Total Length: 84
- Identification: 0x3f4a (16202)
- Flags: 0x02 (Don't Fragment)
- Fragment offset: 0
- Time to live: 64
- Protocol: ICMP (1)
- Header checksum: 0xdf40 [validation disabled]
- Source: 10.0.2.15 (10.0.2.15)
- Destination: 8.8.8.8 (8.8.8.8)
- [Source GeoIP: Unknown]
- [Destination GeoIP: Unknown]

Rysunek 2.6: Wireshark. IP - przykład pakietu.

protokół umieszczony w danych pakietu to ICMP (wartość 0x01). Dalej umieszczony jest adres źródłowy i docelowy, a na końcu adresy te są wykorzystane do określenia lokalizacji źródła i celu, jednakże funkcja ta wymaga zainstalowania w Wiresharku bazy geolokalizacyjnej.

2.3.4 IPSec

TBC

2.4 TCP

Protokoły TCP i UDP są najpopularniejszymi protokołami wykorzystywanyymi w Warstwie Transportowej modelu OSI.

Pierwszy z nich to protokół TCP, będący częścią szeroko wykorzystywanego modelu TCP/IP, którego głównym zadaniem jest wprowadzenie niezawodności (ang. reliability) w komunikacji. Protokół IP z warstwy niższej jest protokołem zawodnym, co znaczy, że nie gwarantuje dostarczenia pakietów do odbiorcy, a ponadto mogą one dotrzeć do odbiorcy w innej kolejności, niż zostały wysłane przez nadawcę. TCP gwarantuje protokołom z warstwy wyższej, że pakiety zostaną dostarczone do odbiorcy w całości i w poprawnej kolejności, a także bez duplikatów.

Protokół TCP jest protokołem działającym w trybie klient-serwer (o trybach powiemy sobie więcej w dalszej części skryptu) i jest protokołem strumieniowym, co znaczy, że z punktu widzenia wyższej warstwy oprogramowania, dane płynące połączeniem TCP należy traktować jako ciąg oktetów.

2.4.1 Format TCP

Na Rysunku 2.3 przedstawiono format pakietu TCP.

Bity	0-6	7-15	16-31
0	Port nadawcy		Port odbiorcy
32	Numer sekwencyjny		
64	Numer potwierdzenia		
96	Długość nagłówka	Flagi	Szerokość okna
128	Suma kontrolna		Wskaźnik priorytetu
160	Opcje		

Tablica 2.3: Format pakietu TCP.

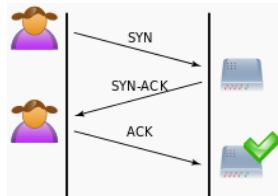
Poszczególne pola mają następującą definicję:

- **Port nadawcy** (16 bitów) - port, z którego strona inicjująca chce nawiązać połączenie.
- **Port odbiorca** (16 bitów) - port, z którym strona inicjująca chce nawiązać połączenie po stronie odbiorcy.
- **Numer sekwencyjny** (32 bity) - w momencie, gdy pakiet jest wysyłany w celu nawiązania połączenia numer ten jest generowany losowo i wykorzystywany do synchronizacji połączenia, zaś gdy flaga SYN nie jest ustawiona (w trakcie dalszej komunikacji) numer sekwencyjny to przesunięcie (ang. offset) przesyłanych w segmencie TCP danych od początku całego (fragmentowanego) pakietu (np. pobieranego pliku).
- **Numer potwierdzenia** (32 bity) - numer będący potwierdzeniem otrzymania segmentu, który jest równy numerowi sekwencyjnemu z otrzymanego segmentu lub numerem o jeden większym, gdy ustawiona jest flaga SYN lub FIN.
- **Długość nagłówka** (4 bity) - długość nagłówka TCP wyrażona w 32-bitowych słowach, jest niezbędne przy określaniu miejsca rozpoczęcia danych.
- **Flagi** (12 bitów) - pierwsze trzy bity są zarezerwowane, a następne to: NS – (ang. Nonce Sum) jednabitowa suma wartości flag ECN (ECN Echo, Congestion Window Reduced, Nonce Sum) weryfikująca ich integralność; CWR – (ang. Congestion Window Reduced) flaga potwierdzająca odebranie powiadomienia przez nadawcę, umożliwia odbiorcy zaprzestanie wysyłania echa; ECE – (ang. ECN-Echo) flaga ustawiana przez odbiorcę w momencie otrzymania pakietu z ustawioną flagą CE; URG – informuje o istotności pola "Priorytet"; ACK – informuje o istotności pola Numer potwierdzenia"; PSH – wymusza przesłanie pakietu; RST – resetuje połączenie (wymagane ponowne uzgodnienie sekwencji); SYN – synchronizuje kolejne numery sekwencyjne; FIN – oznacza zakończenie przekazu danych.

- **Szerokość okna** (16 bitów) - informacja o tym, ile danych może aktualnie przyjąć odbiorca. Wartość 0 wskazuje na oczekiwanie na segment z innym numerem tego pola. Jest to mechanizm zabezpieczający komputer nadawcy przed zbyt dużym napływem danych.
- **Suma kontrolna** (16 bitów) - liczba, będąca wynikiem działań na bitach całego pakietu, pozwalająca na sprawdzenie tego pakietu pod względem poprawności danych. Obliczana jest z całego nagłówka TCP z wyzerowanymi polami sumy kontrolnej oraz ostatnich ośmiu pól nagłówka IP stanowiących adresy nadawcy i odbiorcy pakietu.
- **Wskaźnik priorytetu** (16 bitów) - jeżeli flaga URG jest włączona, informuje o ważności pakietu.
- **Opcje** - dodatkowe informacje i polecenia: 0 – koniec listy opcji, 1 – brak działania, 2 – ustawia maksymalną długość segmentu. W przypadku opcji 2 to tzw. Uzupełnienie, które dopełnia zerami długość segmentu do wielokrotności 32 bitów.

2.4.2 Połączenie TCP

Protokół TCP jest protokołem połączeniowym. Przed wysłaniem danych nawiązywane jest połączenie, które jest później utrzymywane podczas przesyłania danych. Nawiązanie połączenia to procedura zwana *three-way handshake* i umieszczona została na Rysunku 2.7.



Rysunek 2.7: Protokół TCP: Three-way handshake. Źródło: Wikipedia.

Na początku strona, która chce nawiązać połączenie (nazwiemy ją stroną A) wysyła do odbiorcy (strony B) pakiet z ustawioną flagą SYN i uzupełnionym losowym numerem sekwencyjnym (np. 100). Strona B odsyła do nadawcy pakiet z ustawioną flagą SYN oraz ACK (ang. acknowledgment). Wartość numeru sekwencyjnego jest losową wartością (np. 500), wygenerowaną przez stronę A, zaś wartość numeru potwierdzenia jest o jeden większa od otrzymanego numeru sekwencyjnego od strony A (np. 101). Strona A, po otrzymaniu segmentu SYN/ACK wysyła trzeci i ostatni pakiet (stąd nazwa procedury) z ustawioną flagą ACK i wartością numery potwierdzenia o jeden większą od otrzymanego numeru sekwencyjnego od strony B (np. 501). W każdej chwili proces może być przerwany, gdy jedna ze stron wyśle pakiet z ustawioną flagą RST (ang. reset).

Po pomyślnym przeprowadzeniu procesu nawiązywania połączenia obie strony są gotowe na przesyłanie i odbieranie danych. Niezawodność protokołu TCP jest uzyskana m.in. za pomocą sum kontrolnych segmentów, które służą do weryfikacji poprawności pakietu. Ponadto, po każdym wysłanym segmencie strona odbierająca musi odesłać segment z ustawioną flagą ACK i poprawnym numerem potwierdzenia. W sytuacji, gdy strona nadająca nie otrzyma segmentu z potwierdzeniem, wyśle swój segment ponownie. Ponadto, numery sekwencyjne pozwalają stronie odbierającej odtworzyć kolejność wysyłanych segmentów.

W celu zakończenia połączenia jedna ze stron wysyła segment z ustawioną flagą FIN, którego otrzymanie musi zostać potwierdzone (segmentem z flagą ACK). W odpowiedzi, druga strona wysyła segment z flagami FIN/ACK, którego otrzymanie również musi zostać potwierdzone. Po tej 4-krokowej komunikacji połączenie zostaje zakończone.

2.4.3 Stan połączenia

Poniżej przedstawiono możliwe stany połączenia TCP:

- **LISTEN** Gotowość do przyjęcia połączenia na określonym porcie przez serwer. Połączenie TCP może zostać nawiązane tylko wtedy, gdy jedna ze stron (zwana serwerem) oczekuje na chęć nazwiązania z nią połączenia.
- **SYN-SENT** Pierwsza faza nawiązywania połączenia przez stronę inicującą. Wysłano pakiet z flagą SYN. Oczekiwanie na pakiet SYN/ACK.
- **SYN-RECEIVED** Otrzymano pakiet SYN, wysłano SYN/ACK. Trwa oczekiwanie na ACK. Połączenie jest w połowie otwarte (ang. half-open).
- **ESTABLISHED** Połączenie zostało prawidłowo nawiązane. Prawdopodobnie trwa transmisja.
- **FIN-WAIT-1** Wysłano pakiet FIN. Dane wciąż mogą być odbierane ale wysyłanie jest już niemożliwe.
- **FIN-WAIT-2** Otrzymano potwierdzenie własnego pakietu FIN. Oczekuje na przesłanie FIN/ACK od drugiej strony.
- **CLOSE-WAIT** Otrzymano pakiet FIN, wysłano ACK. Oczekiwanie na przesłanie własnego pakietu FIN/ACK (gdy aplikacja skończy nadawanie).
- **CLOSING** Połączenie jest zamykane.
- **LAST-ACK** Otrzymano i wysłano FIN. Trwa oczekiwanie na ostatni pakiet ACK.
- **TIME-WAIT** Oczekiwanie w celu upewnienia się, że druga strona otrzymała potwierdzenie rozłączenia. Zgodnie z RFC 793 połączenie może być w stanie TIME-WAIT najdłużej przez 4 minuty.
- **CLOSED** Połączenie jest zamknięte.

2.4.4 Przykład podsłuchanego pakietu TCP

Na Rysunku 2.8 znajduje się zrzut ekranu z Wiresharka ze zdekodowanym segmentem TCP.

```

5 0.012159000 10.0.2.15 212.77.98.9 TCP 74 51633>80 [SYN] Seq=0 Win=29200 Len=0 MSS=1460
6 0.034590000 212.77.98.9 10.0.2.15 TCP 68 80>51633 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0
7 0.034637800 10.0.2.15 212.77.98.9 TCP 54 51633>80 [ACK] Seq=1 Ack=1 Win=29200 Len=0
8 0.034784000 10.0.2.15 212.77.98.9 HTTP 127 GET / HTTP/1.1

Internet Protocol Version 4, Src: 212.77.98.9 (212.77.98.9), Dst: 10.0.2.15 (10.0.2.15)
Transmission Control Protocol, Src Port: 80 (80), Dst Port: 51633 (51633), Seq: 0, Ack: 1, Len: 0
Source Port: 80 (80)
Destination Port: 51633 (51633)
[Stream index: 0]
[TCP Segment Len: 0]
Sequence number: 0 (relative sequence number)
Acknowledgment number: 1 (relative ack number)
Header Length: 24 bytes
... 0000 0001 0010 = Flags: 0x012 (SYN, ACK)
Window size value: 65535
[Calculated window size: 65535]
Checksum: 0x2556 [validation disabled]
Urgent pointer: 0
Options: (4 bytes), Maximum segment size
[SEQ/ACK analysis]

```

Rysunek 2.8: Segment TCP.

Widać na nim, że segment ten ma ustawione flagi SYN i ACK, czyli jest drugim segmentem procedury 3-way handshake. Został wysłany z adresu 212.77.98.9 (Czy jesteś w stanie powiedzieć, dla jakiej domeny jest to adres?) z portu 80 na adres 10.0.2.15 na port 51633, zatem najprawdopodobniej komputer o adresie 10.0.2.15 próbuje się połączyć z serwerem WWW. Długość nagłówka to 24 bajty, zaś długość danych to 0 bajtów, ponieważ ten segment, to segment kontrolny.

Segment posiada numer sekwencyjny równy 0 oraz numer potwierdzenia równy 1. Są to jednak wartości względne, ponieważ Wireshark prezentuje wartości tych numerów względem ich początkowej wartości. Na przykład, jeśli w segmencie SYN wysłano sumer sekwencyjny równy 500, a w segmencie SYN/ACK wartość (względna) numeru potwierdzenia jest równa 1, to jej wartością faktyczną jest 501. Ponadto, w połączeniu wyłączono validację sumy kontrolnej.

Protokół TCP jest protokołem strumieniowym, dlatego w Wiresharku istnieje opcja, która pozwala połączyć wszystkie przesłane segmenty TCP i wykusić dane przesłane protokołem warstwy aplikacji w ramach jednego połączenia TCP.

2.5 UDP

Protokół UDP (ang. User Datagram Protocol) jest, podobnie jak TCP, protokołem Warstwy Transportowej, jednakże jest on zupełnie inny niż TCP. UDP nie jest protokołem połączniowym, więc nie ma narzutu na nawiązywanie połączenia i śledzenie sesji (w przeciwieństwie do TCP). Nie ma też mechanizmów kontroli przepływu i retransmisji, w związku z czym nie gwarantuje dostarczenia datagramu. Jest to protokół typu *fire-and-forget*, czyli wysyłający pakiet i zapominający o jego istnieniu.

Korzyścią płynącą z takiego uproszczenia budowy jest większa szybkość transmisji danych i brak dodatkowych zadań, którymi musi zajmować się komputer posługujący się tym protokołem. Z tych względów UDP jest często używany w takich zastosowaniach jak wideokonferencje, strumienie dźwięku w Internecie i gry sieciowe, gdzie dane muszą być przesyłane możliwie szybko, a poprawianiem błędów zajmują się inne warstwy modelu OSI. Przykładem może być VoIP lub protokół DNS.

UDP udostępnia mechanizm identyfikacji różnych punktów końcowych (np. pracujących aplikacji, usług czy serwisów) na jednym hoście dzięki portom. UDP zajmuje się dostarczaniem pojedynczych pakietów, udostępnionych przez IP, na którym się opiera. Kolejną cechą odróżniającą UDP od TCP jest możliwość transmisji do kilku adresów docelowych naraz (tzw. multicast).

2.5.1 Format UDP

Na Rysunku 2.4 przedstawiono format pakietu TCP.

Bitы	0-15	16-31
0	Port nadawcy	Port odbiorcy
32	Długość	Suma kontrolna
64		Dane

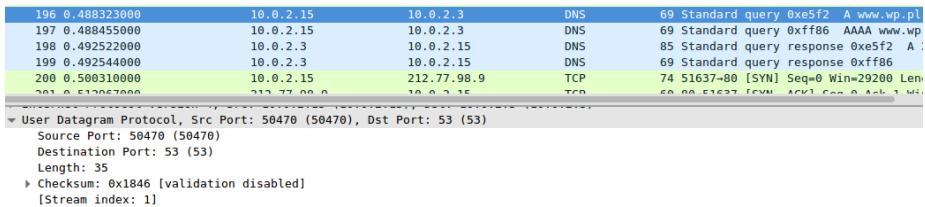
Tablica 2.4: Format pakietu UDP.

Poszczególne pola mają następującą definicję:

- **Port nadawcy** (16 bitów) - port, z którego została wysłana wiadomość. Gdy jest ustawiony może zostać przyjęty jako port, do którego powinna zostać zwrócona wiadomość zwrotna w przypadku braku innej informacji. Port nadawcy jest polem opcjonalnym. Gdy pole to nie jest używane przyjmuje wartość zero.
- **Port odbiorcy** (16 bitów) - port, z którym strona inicjująca chce nawiązać połączenie po stronie odbiorcy.
- **Długość** (16 bitów) - długość w bajtach całego datagramu: nagłówek i dane. Minimalna długość to 8 bajtów i jest to długość nagłówka. Wielkość pola ustala teoretyczny limit 65527 bajtów, dla danych przenoszonych przez pojedynczy datagram UDP.
- **Suma kontrolna** (16 bitów) - pole użyte do sprawdzania poprawności nagłówka oraz danych. Pole jest opcjonalne. Ponieważ IP nie wylicza sumy kontrolnej dla danych (jedynie dla nagłówka), suma kontrolna UDP jest jedyną gwarancją, że dane nie zostały uszkodzone.

2.5.2 Przykład podsłuchanego pakietu UDP

Na Rysunku 2.9 znajduje się zrzut ekranu z Wiresharka ze zdekodowanym segmencitem UDP.



Rysunek 2.9: Segment UDP.

Widać na nim, że segment został wysłany z adresu 10.0.2.15 z portu 50470 na adres 10.0.2.3 na port 53, zatem najprawdopodobniej komputer o adresie 10.0.2.15 próbuje dowiedzieć się, jaki jest adres IP domeny, wykorzystując protokół DNS warstwy wyższej. Długość segmentu to 35 bajtów, a w segmencie wyłączeno validację sumy kontrolnej.

2.6 DNS

Protokół DNS (ang. Domain Name System) to protokół z Warstwy Aplikacji służący do odnalezienia adresu IP skojarzonego z domeną, opisany w dokumencie RFC 1035. Wykorzystuje on zdecentralizowaną bazę danych, która przechowuje pary: adres IP i domena.

DNS to system hierarchiczny i otwarty, do którego należy wiele organizacji, które uruchamiają własne serwery DNS. Jednymi z popularnych serwerów DNS są serwery Google o adresach 8.8.8.8 i 8.8.4.4.

Popularnym narzędziem wykorzystującym protokół DNS jest *nslookup* (dostępny domyślnie na systemach Windows, MacOS i Linux), który po podaniu domeny zwraca jej adres, jeśli domena ta istnieje w bazie systemu DNS. Przykład wykorzystania umieściłem na Rysunku 2.10.

```
root@debian-vm:~# nslookup www.google.pl
Server:          10.0.2.3
Address:         10.0.2.3#53
Non-authoritative answer:
Name:   www.google.pl
Address: 216.58.209.35
```

Rysunek 2.10: Program *nslookup*.

W przykładzie wykorzystałem protokół DNS do określenia adresu IP domeny *www.google.pl*. Jak widać znajduje się ona pod adresem 216.58.209.35.

2.6.1 Format DNS

Ogólny format komunikatu DNS znajduje się poniżej na Rysunku 2.5. Nagłówek jest obecny w każdym komunikacie i określa liczbę pozostałych sekcji.

Nagłówek
Zapytanie do serwera nazw
Odpowiedź z serwera nazw
Zwierzchność - wskazuje serwery zwierzchnie dla domeny
Sekcje dodatkowe

Tablica 2.5: Ogólny format komunikatu DNS.

Nagłówek znajduje się na Rysunku 2.6.

Bit 0	1 - 4	5	6	7	8	9 - 11	12-15
ID							
QR	OP	AA	TC	RD	RA	Z	RC
			Liczba sekcji Zapytanie				
			Liczba sekcji Odpowiedź				
			Liczba sekcji Zwierzchność				
			Liczba sekcji Dodatkowych				

Tablica 2.6: Format nagłówka DNS.

Poszczególne pola mają następującą definicję:

- **ID** - (IDentifier) - identyfikator tworzony przez program wysyłający zapytanie; serwer przepisuje ten identyfikator do swojej odpowiedzi, dzięki czemu możliwe jest jednoznaczne powiązanie zapytania i odpowiedzi.
- **QR** - (Query or Response) - określa, czy komunikat jest zapytaniem (0) czy odpowiedzią (1).
- **OP** - określa rodzaj zapytania wysyłanego od klienta, jest przypisywany przez serwer do odpowiedzi. Wartości:
 - 0 - QUERY - standardowe zapytanie,
 - 1 - IQUERY - zapytanie zwrotne,
 - 2 - STATUS - pytanie o stan serwera,
 - 3-15 - zarezerwowane do przyszłego użytku.
- **AA** - (Authoritative Answer) - oznacza, że odpowiedź jest autorytywna.
- **TC** - (TrunCation) - oznacza, że odpowiedź nie zmieściła się w jednym pakiecie UDP i została obcięta.

- **RD** - (Recursion Desired) - oznacza, że klient żąda rekurencji – pole to jest kopiowane do odpowiedzi.
- **RA** - (Recursion Available) - bit oznaczający, że serwer obsługuje zapytania rekurencyjne.
- **Z** - zarezerwowane do przyszłego wykorzystania. Pole powinno być wyzerowane.
- **RC** - (Response CODE) kod odpowiedzi. Przyjmuje wartości:
 - 0 - brak błędu,
 - 1 - błąd formatu - serwer nie potrafił zinterpretować zapytania,
 - 2 - błąd serwera - wewnętrzny błąd serwera,
 - 3 - błąd nazwy - nazwa domenowa podana w zapytaniu nie istnieje,
 - 4 - nie zaimplementowano - serwer nie obsługuje typu otrzymanego zapytania,
 - 5 - odrzucono - serwer odmawia wykonania określonej operacji, np. transferu strefy,
 - 6-15 - zarezerwowane do przyszłego użytku.

Format sekcji Zapytanie znajduje się na Rysunku 2.7.

QNAME	Odpytywana domena.
QTYPE	Odpytywany rekord domeny.
QCLASS	Klasa odpytywanego rekordu.

Tablica 2.7: Format sekcji Zapytanie.

Poszczególne pola mają następującą definicję:

- **QNAME** - nazwa, o którą odpytywany jest serwer. Nazwa dzielona jest na części (ang. label), gdzie separatorem jest znak kropki (np. www.wp.pl będzie miała 3 części). Przed każdą częścią umieszczony jest bajt, który zawiera liczbę znaków występujących w poprzedzanej części. Dla części drugiej i każdej kolejnej, bajt z długością zastępuje kropkę (np. www.wp.pl zostanie zmienione na 0x03 www 0x02 wp 0x02 pl, gdzie 0x00 to reprezentacja szestastkowa bajtu).
- **QTYPE** (2 bajty) - rekord domeny, o który odpytywany jest serwer. Popularnymi wartościami są: 1 - rekord A, 2 - rekord NS, czy 15 rekord MX. Wszystkie wartości dostępne są pod adresem <http://www.iana.org/assignments/dns-parameters/dns-parameters.xhtml>.
- **QCLASS** (2 bajty) - klasa rekordu, o który odpytywany jest serwer. Najpopularniejszą wartością jest 1, oznaczająca Internet. Wszystkie wartości dostępne są pod adresem <http://www.iana.org/assignments/dns-parameters/dns-parameters.xhtml>.

NAME	Nazwa domeny otrzymana w zapytaniu.
TYPE	Rekord domeny otrzymany w zapytaniu.
CLASS	Klasa odpytywanego rekordu otrzymana w zapytaniu.
TTL	Czas w sekundach, przez który rekord może być zapamiętany.
RDLENGTH	Wielkość odpowiedzi (RDATA) w bajtach.
RDATA	Wartość odpytywanego rekordu.

Tablica 2.8: Format sekcji Odpowiedź.

Format sekcji Odpowiedź znajduje się na Rysunku 2.8.

Poszczególne pola mają następującą definicję:

- **NAME** - pole może być zdefiniowane według jednego z dwóch formatów. Pierwszy format jest dokładnie taki sam, jak pola QNAME w zapytaniu (patrz wyżej). Drugi format o format wskaźnika, w którym wartość ma dokładnie 16 bitów, gdzie dwa pierwsze są jedynkami, zaś pozostałe to indeks (indeksowanie od 0) bajtu rozpoczynającego pole QNAME (w sekcji Zapytanie) względem początku danych DNS (początku nagłówka DNS).
- **TYPE** (2 bajty) - wartość tego pola jest dokładnie taka sama, jak pola QTYPING w zapytaniu (patrz wyżej).
- **CLASS** (2 bajty) - wartość tego pola jest dokładnie taka sama, jak pola QCCLASS w zapytaniu (patrz wyżej).
- **TTL** (4 bajty) - czas w sekundach, przez który rekord może być zapamiętany (ang. cached). Wartość 0 oznacza, że rekord nie może być zapamiętany.
- **RDLENGTH** (2 bajty) - długość pola RDATA w bajtach.
- **RDATA** - wartość odpytywanego rekordu. W zależności od typu odpytywanego rekordu jego wartość może być różna. Dla rekordu A wartością jest 4-bajtowa liczba reprezentująca adres IP.

2.6.2 Przykład podsłuchanej komunikacji DNS

Na Rysunkach 2.11 i 2.12 znajdują się zrzuty ekranu z Wiresharka dla, odpowiednio, zapytania i odpowiedzi DNS. Protokół DNS wykorzystuje protokół UDP do odpytywania o rekordy domeny co widać na Rysunku 2.11 (wpis User Datagram Protocol nad Domain Name System).

Zapytanie ma identyfikator 0xe5f2. Drugie dwa bajty, czyli flagi zapytania mają wartość 0x0100, co oznacza, że tylko flaga RD jest ustawiona, czyli komputer 10.0.2.15, wysyłający zapytanie, oczekuje na odpowiedź zawierającą treść zapytania. Wartość pierwszej flagi QR (pierwszy bit) równa 0 oznacza, że jest to zapytanie.

Wśród pól oznaczających liczebność sekcji tylko dla sekcji Zapytanie ustawiona jest wartość 1, dla pozostałych zaś 0, więc w komunikacie DNS znajduje

106 0.488323000	10.0.2.15	10.0.2.3	DNS	60 Standard query 0xe5f2 A www.wp.pl
197 0.488455000	10.0.2.15	10.0.2.3	DNS	69 Standard query 0xff86 AAAA www.wp.pl
198 0.492522000	10.0.2.3	10.0.2.15	DNS	85 Standard query response 0xe5f2 A 212.77.98.9
199 0.492544000	10.0.2.3	10.0.2.15	DNS	69 Standard query response 0xff86
► Internet Protocol Version 4, Src: 10.0.2.15 (10.0.2.15), Dst: 10.0.2.3 (10.0.2.3)				
► User Datagram Protocol, Src Port: 50470 (50470), Dst Port: 53 (53)				
► Domain Name System (query)				
[Request In: 106]				
Transaction ID: 0xe5f2				
Flags: 0x0100 Standard query				
Questions: 1				
Answer RRs: 0				
Authority RRs: 0				
Additional RRs: 0				
► Queries				
www.wp.pl: type A, class IN				
Name: www.wp.pl				
[Name Length: 9]				
[Label Count: 3]				
Type: A (Host Address) (1)				
Class: IN (0x0001)				

Rysunek 2.11: Zapytanie DNS.

się zapytanie tylko o 1 rekord domeny. Poniżej rozwinięta jest treść sekcji Zapytanie. Widać w niej, że zapytanie dotyczy domeny `www.wp.pl`, która ma 9 znaków i 3 części (label). Odpytywany rekord to `A`, zaś klasa to Internet.

198 0.492522000	10.0.2.3	10.0.2.15	DNS	85 Standard query response 0xe5f2 A 212.77.98.9
199 0.492544000	10.0.2.3	10.0.2.15	DNS	69 Standard query response 0xff86
200 0.506310000	10.0.2.15	212.77.98.9	TCP	74 51637-80 TSYN Se=0 Win=29200 Len=0 MSS=1460
► Domain Name System (response)				
[Request In: 106]				
[Time: 0.004199000 seconds]				
Transaction ID: 0xe5f2				
Flags: 0x8180 Standard query response, No error				
Questions: 1				
Answer RRs: 1				
Authority RRs: 0				
Additional RRs: 0				
► Queries				
► Answers				
www.wp.pl: type A, class IN, addr 212.77.98.9				
Name: www.wp.pl				
Type: A (Host Address) (1)				
Class: IN (0x0001)				
Time to live: 289				
Data length: 4				
Address: 212.77.98.9 (212.77.98.9)				

Rysunek 2.12: Odpowiedź DNS.

W odpowiedzi wartość identyfikatora jest taka sama, jak w zapytaniu (tak są ze sobą kojarzone pary). Wśród flag ustawione są 3 bity: QR, oznaczający odpowiedź; RD jak wyżej oraz RA, czyli informacja, że serwer DNS umie obsługiwać zapytania z prośbą o rekursję (RD).

Jako że odpowiedź jest wygenerowana na prośbę RD to zawiera w sobie sekcję Zapytanie oraz jedną sekcję Odpowiedź. W sekcji Zapytanie wszystkie wartości są takie same jak w zapytaniu z Rysunku 2.11.

W sekcji Odpowiedź wartości pól NAME, TYPE i CLASS są takie same, jak w sekcji Zapytanie, przy czym pole NAME jest zdefiniowane według formatu wskaźnikowego. Wartość tego pola to `0xc00c`. Pierwszy bajt `0xc0` to wymagany przez format prefix, czyli dwa pierwsze bity ustawione na 1. Drugi bajt `0x0c`, to wartość 12 i pod tym indeksem znajduje się początek pola QNAME w sekcji Zapytanie (indeksowanie od zera).

W sekcji Odpowiedź, oprócz pól z zapytaniem, jest również pole TTL równe 289, czyli adres skojarzony z domeną można zapamiętać przez 289 sekund. Następnie ustawione jest pole RDLENGTH równe 4, co znaczy, że wartość odpy-

tywanego rekordu ma 4 bajty. Istotnie, wartością rekordu A jest adres IP, który ma cztery bajty i na Rysunku 2.12 jest równy 212.77.98.9.

3

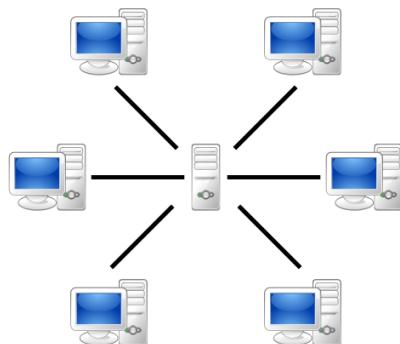
Architektury sieci

Rozdział ten najlepiej rozpocząć od pytania, czym różni się pobieranie pliku ze strony WWW od pobierania pliku z tzw. torrentów? Efekt końcowy jest taki sam, jednak sam proces jest inny.

Przede wszystkim pliki te pobierane są w ramach całkowicie różnych architektur sieciowych. W pierwszym przypadku łączymy się z serwerem i z niego pobieramy cały plik, zaś w przypadku torrentów pobieramy plik z wielu źródeł po kawałku (ale równocześnie). Te dwie różne architektury omówimy w tym rozdziale.

3.0.1 Klient-Serwer

Architektura Klient-Serwer to najpopularniejsza architektura w świecie Internetu. Zwykle mówimy, że łączymy się z jakimś serwerem, a dokładniej wiele komputerów łączy się z serwerem, czy to WWW, pocztowym, czy FTP.



Rysunek 3.1: Architektura Klient-Serwer.
Źródło: Wikipedia.pl

Schemat architektury Klient-Serwer przedstawiono na Rysunku 3.1. Serwer znajduje się w środku, zaś dookoła są Klienci. Architektura ta nazywana jest scentralizowaną, ponieważ wszyscy łączą się z centralnym serwerem.

Serwer to komputer, na którym uruchomiono aplikację serwerową, która dostarcza pewne zasoby (np. strony WWW). Klient to odbiorca tego zasobu (np. przeglądarka internetowa).

Zwykle serwer i klient uruchomione są na różnych komputerach, bo zwykle chcemy pobrać zasób, którego nie posiadamy, ale nic nie stoi na przeszkodzie, żeby zarówno serwer, jak i klient były uruchomione na jednej maszynie.

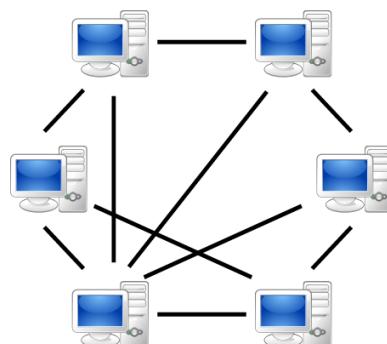
Z drugiej strony często komputer jest zarówno serwerem i klientem, ale dla innych usług. Na przykład maszyna, z którą łączymy się przez RDP jest serwerem RDP, a jeśli na niej uruchamiamy przeglądarkę internetową jest również klientem WWW. Krótko mówiąc, dany komputer można nazwać jednoznacznie serwerem lub klientem tylko w kontekście pewnego protokołu wymiany danych.

Komunikację w architekturze rozpoczyna Klient, ponieważ to on chce uzyskać pewien zasób. Obie strony muszą *rozmawiać* wspólnym językiem oraz postępować według określonych reguł, dzięki którym będą rozumiały wzajemne komunikaty. Język oraz zestaw reguł określa protokół.

Sposób dalszej komunikacji z serwerem może być różny. Najpopularniejsza metoda to Zapytanie-Odpowiedź (ang. Request-Response). Oprócz niej istnieją inne metody, np. Publikuj-Subskrybuj (ang. Publish-Subscribe). Obie zostaną omówione dokładniej w dalszej części rozdziału.

3.0.2 Peer-2-Peer

Podstawową cechą architektury Peer-to-peer (P2P) jest jej decentralizacja. W niej nie ma rozdzielenia na serwery i klientów, lecz wszystkie komputery są równorzędne i stąd ich nazwa - peer. Schemat architektury P2P przedstawiono na Rysunku 3.2.

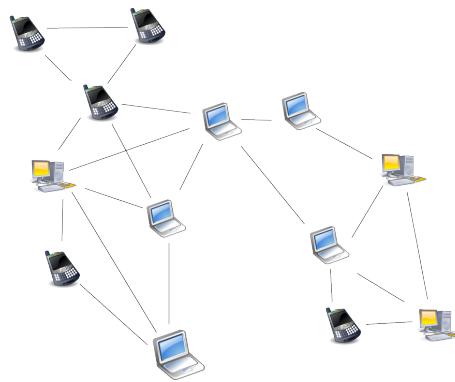


Rysunek 3.2: Architektura Peer2Peer.
 Źródło: [Wikipedia.pl](https://pl.wikipedia.org)

Z historycznego punktu widzenia architektura P2P stała się popularna za sprawą systemów współdzielenia plików (Napster) mimo, że wykorzystywana była dużo wcześniej.

Sieć P2P to sieć równoważnych komputerów, z których każdy jest zarówno serwerem (dostarcza zasoby), jak i klientem (pobiera zasoby). W przypadku modelu Klient-Serwer jedna maszyna dostarczała zasoby.

Sieci P2P są opisywane jako wirtualne sieci tworzone przez komunikujących się ze sobą użytkowników, niezależne od sieci fizycznych, które nie są podporządkowane żadnym władzom. Stąd ich popularność na rynku pirackim - łatwiej usunąć plik lub wyłączyć jeden serwer, niż tysiące, czy setki tysięcy peerów. Komunikacja między peerami odbywa się w ramach sieci fizycznych (TCP/IP) jednakże sieć P2P formuje warstwę nadzrędną, która może mieć swoją strukturę lub może istnieć bez określonej struktury.



Rysunek 3.3: Sieć Peer2Peer bez struktury.

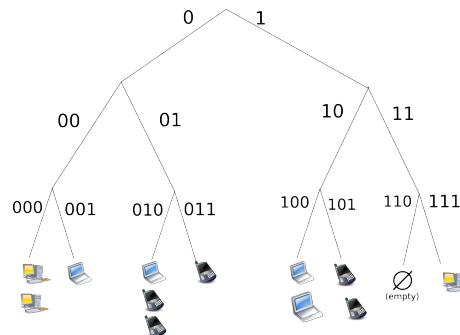
Źródło: [Wikipedia.pl](https://pl.wikipedia.org)

Plusem sieci bez struktury jest łatwa jej budowa, czyli dołączanie nowych węzłów. Wystarczy, że znamy adres chociaż jednego węzła z sieci. Dzięki temu taka sieć jest odporna na częste dołączanie się i odłączanie się węzłów.

Z kolei sporym ograniczeniem jest to, że gdy nowy węzeł w sieci chce znaleźć jakiś zasób, musi odpytać wszystkie pozostałe węzły (lub ich podzbior, ale wtedy znajdzie mniej węzłów z poszukiwanym zasobem), czy posiadają dany zasób. Takie przeszukiwanie sieci znacznie zwiększa wykorzystanie CPU oraz ruch sieciowy. Dodatkowo, nie ma gwarancji, że znajdziemy chociaż jeden węzeł z zasobem, a przeszukiwanie i tak musimy wykonać.

Celem wprowadzenia struktury do sieci P2P jest umożliwienie efektywnego przeszukiwania sieci pod kątem zasobów. Najpopularniejszym typem struktury jest DHT (ang. distributed hash table), która przypisuje zasób do wszystkich peerów, które go posiadają. Idea polega na obliczeniu skrótu danych (np. nazwy pliku), który jest wykorzystywany jako klucz w tabeli. Wartością pod danym kluczem jest lista peerów, które posiadają plik odpowiadający skrótwi.

Jednakże ograniczeniem ustrukturyzowanej sieci jest zarządzanie DHT, które



Rysunek 3.4: Sieć Peer2Peer ze strukturą DHT.
Źródło: Wikipedia.pl

leży w gestii węzłów. Jednym z pomysłów jest rozdzielenie kluczy pomiędzy węzły tak, żeby każdy z nich posiadał *bliskie* sobie klucze (są różne miary określania odległości). Wtedy, nowy węzeł oblicza skrót każdego z plików i wysyła do sieci wiadomość z kluczem, która jest przekazywana, aż dotrze do węzła odpowiedzialnego za dany klucz. Gdy inny węzeł szuka jakiegoś pliku, oblicza skrót jego nazwy i pyta dowolny węzeł o klucz. Zapytanie jest przekazywane, aż do trze do węzła odpowiedzialnego za ten klucz, który w odpowiedzi zwraca listę peerów zawierających ten plik.

Ze względu na złożony proces dołączania i odłączania węzłów z sieci, sieć ustrukturyzowana nie jest odporna na częste dołączanie się i odłączanie węzłów.

Popularnym zastosowaniem sieci P2P (bez lub ze strukturą) jest jej współistnienie z centralnym serwerem, działającym w modelu Klient-Serwer. Taka hybryda cechuje się dużą wydajnością. Centralny serwer jest odpowiedzialny za przechowywanie listy dołączonych węzłów do sieci oraz listę plików z przypisanymi węzłami, które te pliki zawierają. Dzięki centralnemu serwerowi, przeszukiwanie jest dużo wydajniejsze.

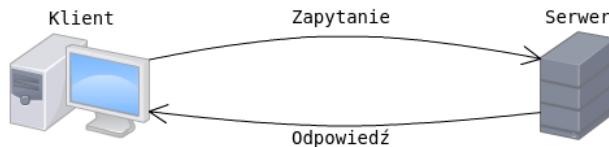
Sieć WWW, w oczach jego twórcy Tima Bernersa-Lee, miała być siecią P2P, gdzie każdy użytkownik miał ją współtworzyć dodając treści oraz linki między stronami.

3.1 Metody komunikacji

Oprócz architektury sieci, w której znajdują się komunikujące się ze sobą komputery istotna jest również metoda komunikacji. Podczas gdy najpopularniejszą jest Request-Response, istnieją również inne, np. Publish-Subscribe. W tym rozdziale omówię obie z nich.

3.1.1 Request-response

Wzorzec Zapytanie-Odpowiedź to najprostsza i najpopularniejsza metoda komunikacji, której schemat przedstawiono na Rysunku 3.5. Klasycznym przykładem są serwery WWW, SSH i wiele innych.



Rysunek 3.5: Wzorzec Zapytanie-Odpowiedź.

Komunikacja jest rozpoczynana przez klienta, który wysyła zapytanie o zasób do serwera. Zadaniem serwera jest przetworzenie zapytania i zwrócenie odpowiedzi, która albo zawiera zasób albo informację, dlaczego zasobu nie można zwrócić. Ten rodzaj komunikacji jest synchroniczny, ponieważ serwer czeka, dopóki nie skontaktuje się z nim klient.

Istnieje częściowo asynchroniczna wersja wzorca Zapytanie-Odpowiedź, w której odpowiedź jest co prawda wysyłana po zapytaniu, ale nie jest określony czas, po którym zostanie ona wysłana. Taka sytuacja może wystąpić, gdy na przykład do przetworzenia zapytania wymagana jest ingerencja człowieka. Taki rodzaj komunikacji jest określany mianem *sync over async*.

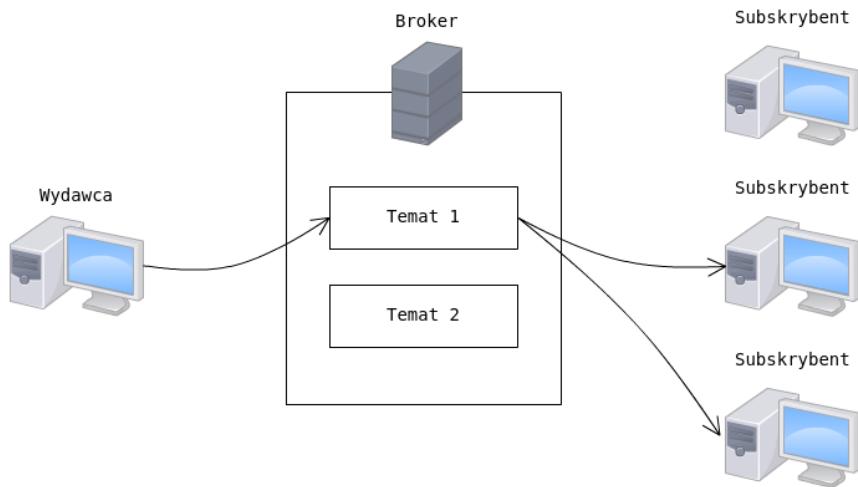
3.1.2 Publish-subscribe

Wzorzec komunikacji Opublikuj-Subskrybuj jest zgoła inny od Zapytanie-Odpowiedź. Spójrzmy najpierw na jego schemat na Rysunku 3.6.

W tym wzorcu pojawiają się 3 role. Pierwsza z nich to Wydawca, który publikuje wiadomości. Wiadomości przekazywane są do Brokeru, który przekierowuje je do wybranych Subskrybentów. Subskrybenci natomiast są odbiorcami tych wiadomości.

Pierwsza, zasadnicza różnica to fakt, że Wydawca wysyła wiadomości do konkretnego Subskrybenta, lecz w tzw. eter. Jego zadanie to po prostu opublikowanie wiadomości. Subskrybent natomiast otrzymuje wiadomości, ale nie wtedy gdy sam o nie poprosi, lecz wtedy gdy Wydawca je opublikuje. Jest to zatem rodzaj komunikacji asynchronicznej lub inaczej mówiąc zdarzeniowej, ponieważ wydawca generuje zdarzenie opublikowania wiadomości, a subskrybent to zdarzenie obsługuje.

Subskrybent zwykle nie otrzymuje wszystkich opublikowanych wiadomości, lecz ich podzbiór, w zależności od typu wiadomości. Subskrybent zgłasza się po wiadomości konkretnego typu i jeśli opublikowana wiadomość jest tego typu,



Rysunek 3.6: Wzorzec Opublikuj-Subskrybij.

to jest przekazywana do Subskrybenta oraz wszystkich pozostałych subskrybentów, którzy zgłosili się po wiadomości tego typu. Zwykle w systemach realizujących ten wzorzec wprowadza się Brokera, który odpowiedzialny jest za filtrowanie wiadomości. Wtedy, jak pokazano na Rysunku 3.6 Wydawca przesyła wiadomości do Brokera, a jego zadaniem jest przekierowanie ich dalej do tych Subskrybentów, którzy się u niego na wiadomości tego typu zapisali.

Zaletą tego wzorca jest fakt, że Wydawca i Subskrybenci nie są ze sobą bezpośrednio związane, a dodanie nowego lub usunięcie Subskrybenta jest operacją przezroczystą dla Wydawcy. We wzorcu Klient-Serwer obie strony muszą jednocześnie być aktywne, aby mogły się ze sobą komunikować. Natomiast wadą wzorca jest konsekwencja braku bezpośredniego powiązania Wydawcy i Subskrybenta, która polega na tym, że Wydawca może zakładać, że jakiś Subskrybent oczekuje na wiadomości, podczas gdy takiego Subskrybenta nie ma.

Przykładem wykorzystania wzorca Publish-Subscribe są graficzne interfejsy użytkownika, w których Subskrybentami są funkcje, które reagują na zdarzenia, np. wciśnięcie przycisku.

4

Programowanie aplikacji sieciowych

W rozdziale dotyczącym programowania aplikacji sieciowych na początku omówię podstawy komunikacji między procesami i komputerami za pomocą gniazd, po czym przedstawię prosty przykład. Będzie to wstęp do pierwszych, prostych zadań. Następnie omówimy sobie mniej znane protokoły oraz sposoby na analizę ich ruchu, w celu napisania klienta lub serwera dla tego protokołu. Na końcu omówimy bardziej zaawansowane mechanizmy wykorzystywane w programowaniu aplikacji sieciowych na poziomie gniazd, a także poznamy kilka bibliotek, które ułatwiają tworzenie aplikacji sieciowych.

Głównym językiem programowania, który będzie wykorzystywany w skrypcie jest Python ze względu na swoją czytelność oraz prostotę. Jednakże większość przykładów dotyczących podstaw będzie również prezentowana w języku C.

Przykłady wykorzystujące dodatkowe biblioteki (np. do komunikacji za pomocą protokołów SMPT, POP, czy HTTP) będą wykorzystywały język Python.

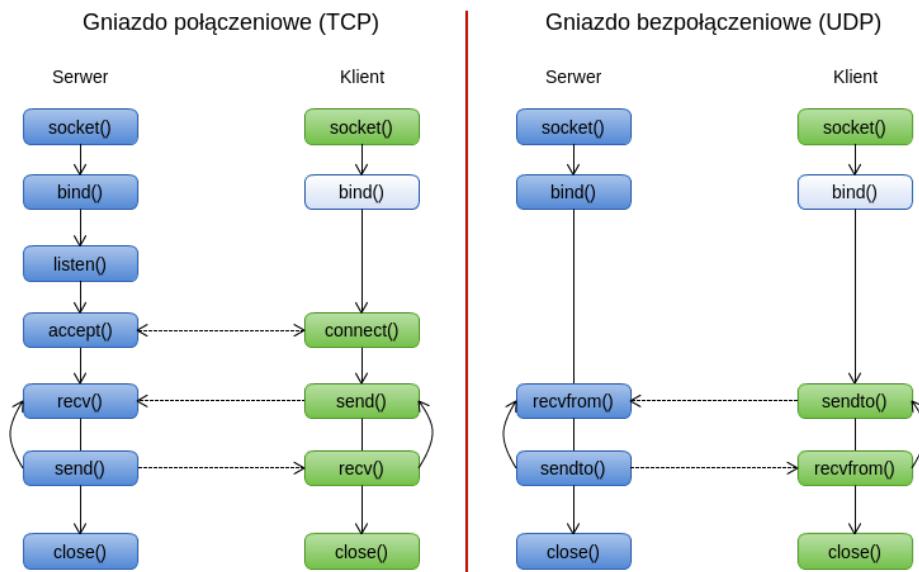
4.1 Gniazda

Programowanie aplikacji sieciowych (bez wykorzystania bibliotek implementujących protokoły Warstwy Aplikacji) rozpoczyna się od gniazd (ang. sockets). Krótko mówiąc gniazda to wirtualne interfejsy, za pomocą których możliwa jest komunikacja między procesami (również tymi, które działają na różnych komputerach).

Zwykle przyjmuje się, że komputer, który nawiązuje połączenie to klient, a komputer, z którym nawiązywane jest połączenie to serwer.

Na początku przedstawię na przykładzie Pythona diagram wywołania metod

(Rysunek 4.1), które składają się na obsługę komunikacji za pomocą gniazd. Jest on bardzo zbliżony do języka C.



Rysunek 4.1: Diagram wywołania metod do obsługi gniazd. Źródło: [7]

Rozgraniczenie na dwa typy komputerów, czyli klient i serwer jest istotne, ponieważ ich działania są różne. Konkretniej to serwer musi wykonać trochę więcej pracy. Po stworzeniu gniazda, serwer musi powiązać (ang. bind) je z konkretnym adresem IP oraz portem.

Jak wspomniałem w opisie protokołów TCP i UDP, w ich formacie znajduje się port, czyli numer 16-bitowy. Dzięki różnym portom, z jednym komputerem może teoretycznie być równocześnie powiązanych 2^{16} gniazd.

Gniazda, podobnie jak protokoły Warstwy Transportowej, możemy podzielić na połączniowe i bezpołączniowe. W przypadku gniazd połączniowych, po powiązaniu gniazda z adresem i portem może ono rozpoczęć nasłuchiwanie (ang. listen) na przychodzące połączenia. W celu nawiązania połączenia serwer musi się zgodzić (ang. accept) na przychodzące połączenie i gdy to zrobi połączenie jest uznane za nawiązane (ang. established). W tym momencie obie strony uruchamiają tzw. pętlę wysyłania i nasłuchiwanego (ang. send/receive), np. klient wysyła (ang. send) dane do serwera, serwer je odbiera (ang. receive), po czym odsyła (ang. send) do klienta odpowiedź, i tak w kółko. Kiedy jedna ze stron chce zakończyć połączenie, zamyka (ang. close) gniazdo.

W przypadku gniazd bezpołączniowych (UDP), serwer nie nasłuchiwa na połączenie i nie akceptuje go, lecz od razu oczekuje na dane.

Oczywiście powyższy opis jest bardzo uproszczony, ponieważ prawdziwe serwery wykorzystują dodatkowe (pominięte w powyższym opisie) mechanizmy, jak chociażby wielowątkowość, czy ogólnie równoległa obsługa wielu klientów.

4.1.1 Metody

W tym podrozdziale opiszę każdą z metod wymienionych na Rysunku 4.1 wraz z deklaracją. W następnym podrozdziale przedstawię przykład prostej komunikacji między dwoma procesami, wykorzystujący wszystkie opisane w tym podrozdziale metody.

W języku Python wszystkie metody do obsługi gniazda znajdują się w module `socket`.

W języku C wszystkie funkcje są zadeklarowane w pliku nagłówkowym `sys/socket.h`, jednakże niektóre elementy (np. wartości stałe) występują w innych plikach nagłówkowych. Przy każdej funkcji będą wskazane wszystkie pliki nagłówkowe.

Metoda `socket()`

Metoda `socket()` tworzy obiekt gniazda (w C jest to dekryptor pliku reprezentującego gniazdo), który następnie jest wykorzystywany podczas wywoływania kolejnych metod.

Język Python

```
1 socket.socket([socket_family[, socket_type[, protocol]]])
2
3 # Przykład:
4 # import socket
5 # s = socket.socket()
```

Listing 4.1: Metoda `socket()` w Pythonie.

Parametry:

- **socket_family** - opcjonalny parametr definiuje rodzinę protokołów wykorzystywanych przez gniazdo. Wartością parametru jest jedna ze stałych zdefiniowanych w module `socket`, rozpoczynających się od znaków `AF_`. Przykładowe wartości:

- `AF_INET` (wartość domyślna) - protokół IP w wersji 4,
- `AF_INET6` - protokół IP w wersji 6,
- `AF_UNIX` - reprezentuje typ gniazda, który jest skojarzony z plikiem systemowym. Wykorzystywany do komunikacji między dwoma procesami, gdy znajdują się na tym samym komputerze.

- **socket_type** - opcjonalny parametr definiujący rodzaj gniazda. Wartością parametru jest jedna ze stałych zdefiniowanych w module `socket`, rozpoczynających się od znaków `SOCK_`. Przykładowe wartości:
 - `SOCK_STREAM` (wartość domyślna) - reprezentuje połączenie strumieniowe (TCP),
 - `SOCK_DGRAM` - reprezentuje połączenie typu fire-and-forget (UDP).
- **protocol** - opcjonalny parametr określający protokół wykorzystywany przez gniazdo. Wartością parametru jest jedna ze stałych zdefiniowanych w module `socket`. Zwykle pomijany, ponieważ w takiej sytuacji protokół gniazda jest określany na podstawie parametru `socket_type`. Nie wszystkie dostępne wartości dla `protocol` współgrają z wartościami poprzednich parametrów (np. `IPPROTO_IPV6` nie jest dostępne, gdy wybrana zostanie rodzina `AF_INET`).

Metoda zwraca obiekt gniazda, na którym następnie będą wywoływanie metody. W przypadku niepowodzenia zwraca wyjątek.

Język C

```

1 #include <sys/types.h>
2 #include <sys/socket.h>
3
4 int socket(int domain, int type, int protocol);

```

Listing 4.2: Funkcja `socket()` w C.

Parametry:

- **domain** - parametr definiuje rodzinę protokołów wykorzystywanych przez gniazdo. Wartością parametru jest jedna ze stałych zdefiniowanych w pliku `socket.h`, rozpoczynających się od znaków `AF_`. Przykładowe wartości:
 - `AF_INET` - protokół IP w wersji 4,
 - `AF_INET6` - protokół IP w wersji 6,
 - `AF_UNIX` - reprezentuje typ gniazda, który jest skojarzony z plikiem systemowym. Wykorzystywany do komunikacji między dwoma procesami, gdy znajdują się na tym samym komputerze.
- **type** - parametr definiujący rodzaj gniazda. Wartością parametru jest jedna ze stałych zdefiniowanych w pliku `socket.h`, rozpoczynających się od znaków `SOCK_`. Przykładowe wartości:
 - `SOCK_STREAM` (wartość domyślna) - reprezentuje połączenie strumieniowe (TCP),
 - `SOCK_DGRAM` - reprezentuje połączenie typu fire-and-forget (UDP).

- **protocol** - parametr określający protokół wykorzystywany przez gniazdo. Wartością parametru jest numer protokołu (lista dostępna tutaj). Zwykle ustawiany na 0, ponieważ w takiej sytuacji protokół gniazda jest określany na podstawie parametru **type**. Nie wszystkie dostępne wartości dla **protocol** współgrają z wartościami poprzednich parametrów (np. wartość 41 (IPv6) nie jest dostępna, gdy wybrana zostanie rodzina AF_INET).

Funkcja zwraca numer deskryptora reprezentującego gniazdo, który będzie wykorzystywany z pozostałych funkcjach jako wartość parametru. W przypadku błędu funkcja zwraca wartość -1.

Metoda **bind()**

Samo gniazdo to jedynie wartość, w której określono sposób komunikacji (protokoly). Kolejnym krokiem jest skojarzenie gniazd z adresem IP oraz portem, z których ma korzystać podczas połączenia. Właśnie do tego służy metoda **bind()**. Po stronie serwera jest ona wymagana, ponieważ serwer, którego zadaniem będzie oczekiwanie na połączenie musi wiedzieć na jakim adresie (np. lokalnym, wewnętrznej sieci, czy sieci Internet) oraz porcie ma nasłuchiwać.

Skojarzenia gniazda z portem po stronie klienta jest opcjonalne, ponieważ jeśli tego nie zrobimy, to poczas wywołania metody **connect()** (opisanej dalej) komputer sam wybierze jeden z wolnych portów. Jednak nie stoi nic na przeszkodzie, żeby po stronie klienta również zaznaczyć, z którego portu chcemy się łączyć.

Zakres portów, które można wykorzystać do skojarzenia gniazda to porty o numerach większych od 1024 oraz mniejszych od 65536 (2^{16}). Porty poniżej 1024 to tzw. porty zarezerwowane i można z nich korzystać tylko wtedy, gdy posiada się uprawnienia administratora (roota). Górną granicą wynika z faktu, że numer portu to wartość 16-bitowa.

Język Python

```

1 socket.bind(address)
2
3 # Przykład:
4 # import socket
5 # s = socket.socket()
6 # s.bind(('127.0.0.1', 8080))

```

Listing 4.3: Metoda bind() w Pythonie.

Parametry:

- **address** - parametr definiuje adres oraz port, z którym gniazdo ma być skojarzone. Format parametru jest zależny od rodziny protokołów gniazda. Dla rodziny AF_INET jest to dwójka zawierająca adres IP (jako napis) oraz port (jaki liczbę) (np. adres = ('127.0.0.1', 8080)). Adresowi IP można przypisać pusty napis, zaś portowi wartość 0, co, zgodnie z dokumentacją, oznacza, że realne wartości tych parametrów zostaną określone przez system operacyjny (np. zostanie wybrany losowy, nieużywany port).

Metoda nic nie zwraca, a jedynie zmienia stan obiektu gniazda, na rzecz którego została wywołana. W przypadku niepowodzenia (np. port zajęty) zwraca wyjątek.

Język C

```

1 #include <sys/socket.h>
2 /* sa_family_t to unsigned integer */
3
4 struct sockaddr {
5     sa_family_t sa_family;
6     char        sa_data[14];
7 }
8
9
10 int bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen);

```

Listing 4.4: Funkcja bind() w C.

Parametry:

- **sockfd** - deskryptor gniazda zwrócony przez funkcję socket().
- **addr** - adres struktury sockaddr, w której w polu sa_family określona jest rodzina protokołów (np. AF_INET), zaś w polu sa_data ustalony jest adres do skojarzenia, którego format jest zależy od rodziny protokołów. Na przykład dla rodziny AF_INET pole sa_data zawiera port (pierwsze dwa bajty) oraz adres (kolejne 4 bajty) - pozostałe bajty są pomijane.
- **addrlen** - rozmiar struktury addr.

Bezpośrednie wykorzystanie struktury sockaddr jest problematyczne ze względu na format przechowywanych danych, dlatego można skorzystać ze struktury sockaddr_in znajdującej się na listingu 4.5.

```

1 #include <netinet/in.h>
2
3 struct sockaddr_in {
4     short      sin_family;    // e.g. AF_INET
5     unsigned short sin_port;   // e.g. htons(3490)
6     struct in_addr sin_addr;  // see struct in_addr, below
7     char        sin_zero[8];   // zero this if you want to
8 };
9
10 struct in_addr {
11     unsigned long s_addr;    // load with inet_aton()
12 };

```

Listing 4.5: Struktura sockaddr_in.

Idea polega na zastąpieniu struktury sockaddr strukturą sockaddr_in, ponieważ w tej drugiej jest łatwiejszy dostęp do pól reprezentujących port oraz adres IP. Jednakże, żeby nie było zbyt kolorowo, gdy korzystamy z sockaddr_in musimy pamiętać o dwóch rzeczach. O pierwszej to właściwie trzeba pamiętać również jak korzystamy z sockaddr, a chodzi o to, żeby użyć funkcji htons do ustalenia wartości pola sin_port (linia 8 na 4.6) oraz funkcji inet_aton

do ustalenia wartości pola `s_addr` struktury znajdującej się w polu `sin_addr` (linia 9 na 4.6). Funkcje te konwertują liczby (port) oraz adresy IP w formie napisu z formatu używanego przez komputer na format używany w sieci (standardyzacja).

Druga rzecz, o której należy pamiętać to rzutowanie typu struktury `sockaddr_in` na typ oczekiwany przez funkcję `bind`, czyli `sockaddr` (linia 12 na 4.6).

```

1 #include <netinet/in.h>
2 #include <sys/socket.h>
3
4 struct sockaddr_in myaddr;
5 int s;
6
7 myaddr.sin_family = AF_INET;
8 myaddr.sin_port = htons(8080);
9 inet_aton("127.0.0.1", &myaddr.sin_addr.s_addr);
10
11 s = socket(AF_INET, SOCK_STREAM, 0);
12 bind(s, (struct sockaddr*)myaddr, sizeof(myaddr));

```

Listing 4.6: Struktura `sockaddr_in` - przykład.

W języku C również można określić wartość adresu IP i portu, która spowoduje, że system operacyjny sam ustali ich wartości realne. Dla adresu IP jest to stała `INADDR_ANY`, którą również dla bezpieczeństwa warto przekazać jako argument do funkcji `htons()`, mimo że zwykle jej wartością jest 0. Odpowiadającą wartością dla portu jest 0.

Funkcja zwraca wartość 0, jeśli udało się powiązać gniazdo z adresem i portem lub -1 w przypadku błędu.

Metoda `listen()`

Metoda wykorzystywana przez gniazda połączeniowe, czyli takie, których typ jest równy `SOCK_STREAM`. Po skojarzeniu gniazda z adresem i portem, przez serwer wywoływana jest metoda `listen()` w celu rozpoczęcia procesu nasłuchiwanego na przychodzące połączenia.

Zadaniem tej metody jest również określenie maksymalnej liczby oczekujących połączeń. Połączenia oczekujące to takie, które nadeszły, ale nie zostały przyjęte za pomocą metody `accept()` (omówiona dalej).

Po wywołaniu metody `listen()` możesz sprawdzić, czy Twój komputer faktycznie rozpoczął nasłuchiwanie na wybranym porcie za pomocą polecenia `netstat -tln` (w systemie Linux).

Język Python

```

1 socket.listen(backlog)
2
3 # Pzykład:
4 # import socket
5 # s = socket.socket()
6 # ...
7 # s.listen(10)

```

Listing 4.7: Metoda `listen()` w Pythonie.

Parametry:

- **backlog** - parametr określa maksymalną liczbę oczekujących połączeń. Zwykle jest to wartość z zakresu 5-10, a jej górnym ograniczeniem w przypadku systemów Linux jest wartość z pliku /proc/sys/net/core/somaxconn (w moim przypadku 128).

Metoda nic nie zwraca, a jedynie zmienia stan obiektu gniazda, na rzecz którego została wywołana. W przypadku niepowodzenia (np. port zajęty) zwraca wyjątek.

Język C

```
1 #include <sys/socket.h>
2
3 int listen(int sockfd, int backlog);
```

Listing 4.8: Funkcja listen() w C.

Parametry:

- **sockfd** - deskryptor gniazda zwrócony przez funkcję socket().
- **backlog** - parametr określa maksymalną liczbę oczekujących połączeń. Zwykle jest to wartość z zakresu 5-10, a jej górnym ograniczeniem w przypadku systemów Linux jest wartość z pliku /proc/sys/net/core/somaxconn (w moim przypadku 128).

Funkcja zwraca wartość 0, jeśli udało się rozpoczęć nasłuchiwanie lub -1 w przypadku błędu.

Metoda connect()

Metoda connect() służy, jak sama nazwa wskazuje, do nawiązania połączenia z serwerem wykorzystującym gniazdo strumieniowe. Jako parametr przyjmuje adres oraz port serwera, z którym chce się połączyć.

Język Python

```
1 socket.connect(address)
2
3 # Pzykład:
4 # import socket
5 # s = socket.socket()
6 # ...
7 # s.connect(('127.0.0.1', 8080))
```

Listing 4.9: Metoda connect() w Pythonie.

Parametry:

- **address** - parametr definiuje adres oraz port serwera, z którym gniazdo chce się połączyć. Podobnie jak w przypadku metody bind(), format parametru jest zależny od rodziny protokołów gniazda. Dla rodziny AF_INET jest to dwójka zawierająca adres IP (jako napis) oraz port (jaki liczbę) (np. adres = ('127.0.0.1', 8080)).

Metoda nic nie zwraca, a jedynie zmienia stan obiektu gniazda, na rzecz którego została wywołana, która polega na nawiązaniu połączenia (np. wymiany segmentów 3-way Handshake w przypadku protokołu TCP). Wcześniej jednak metoda czeka, aż serwer zaakceptuje połączenie, więc w zależności od zachowania serwera czas wywołania metody `connect()` może być różny.

Metoda zwraca wyjątek, gdy wystąpi błąd.

Język C

```
1 #include <sys/socket.h>
2
3 int connect(int sockfd, struct sockaddr *serv_addr, int addrlen);
```

Listing 4.10: Funkcja `connect()` w C.

Parametry:

- **sockfd** - deskryptor gniazda zwrócony przez funkcję `socket()`.
- **serv_addr** - adres struktury `sockaddr`, w której w polu `sa_family` określona jest rodzina protokołów (np. `AF_INET`), zaś w polu `sa_data` ustalony jest adres serwera, którego format jest zależy od rodziny protokołów. Na przykład dla rodziny `AF_INET` pole `sa_data` zawiera port (pierwsze dwa bajty) oraz adres (kolejne 4 bajty) - pozostałe bajty są pomijane.
- **addrlen** - rozmiar struktury `serv_addr`.

Bezpośrednie wykorzystanie struktury `sockaddr` jest problematyczne ze względu na format przechowywanych danych, dlatego, podobnie jak w przypadku funkcji `bind()`, można skorzystać ze struktury `sockaddr_in` znajdującej się na listingu 4.5.

Na listingu 4.11 znajduje się przykład wykorzystania funkcji `connect()` ze strukturą `sockaddr_in`. Jak widać, jedyna różnica względem funkcji `bind()` (Listing 4.6) to wywołanie funkcji `connect()` zamiast `bind()`.

```
1 #include <netinet/in.h>
2 #include <sys/socket.h>
3
4 struct sockaddr_in myaddr;
5 int s;
6
7 myaddr.sin_family = AF_INET;
8 myaddr.sin_port = htons(8080);
9 inet_aton("127.0.0.1", &myaddr.sin_addr.s_addr);
10
11 s = socket(AF_INET, SOCK_STREAM, 0);
12 connect(s, (struct sockaddr*)myaddr, sizeof(myaddr));
```

Listing 4.11: Funkcja `connect()` w C - przykład.

Funkcja zwraca wartość 0, jeśli udało się nawiązać połączenie lub -1 w przypadku błędu.

Metoda `accept()`

Metoda `accept()` wykorzystywana jest przez serwerowe gniazdo połączeniowe, a jej zadaniem jest przyjęcie oczekującego połączenia. Oczywiście przed jej użyciem należy wywołać metodę `listen()`, żeby serwer zaczął na połączenia nasłuchiwać.

Metoda `accept()` będzie czekała, dopóki nie pojawi się jakieś połączenie, czyli jakiś komputer (lub ten sam komputer) nie wywoła metody `connect()` z adresem IP naszej maszyny oraz portem, na którym nasłuchuje nasz serwer.

Język Python

```

1 socket.accept()
2
3 # Przykład:
4 # import socket
5 # s = socket.socket()
6 # ...
7 # conn, address = s.accept()

```

Listing 4.12: Metoda `accept()` w Pythonie.

Metoda `accept()` jest bezparametrowa i zwraca dwójkę: połączenie oraz adres, z którego przyszło połączenie. Adres ma taki sam format, jak parametr metody `bind()`, np. ('127.0.0.1', 56443). Połączenie jest natomiast nowym gniazdem, na którym można wywoływać już metody do przesyłania danych, opisane w dalszej kolejności. Stare gniazdo natomiast kontynuuje nasłuchiwanie na nowe połączenia.

Metoda wyrzuca wyjątek, gdy wystąpi błąd.

Język C

```

1 #include <sys/socket.h>
2
3 int accept(int sockfd, void *addr, int *addrlen);

```

Listing 4.13: Funkcja `accept()` w C.

Parametry:

- **sockfd** - deskryptor gniazda zwrócony przez funkcję `socket()`.
- **addr** (parametr wyjściowy) - adres struktury `sockaddr` lub `sockaddr_in`, która zostanie uzupełniona adresem oraz portem komputera, z którego nadeszło połączenie.
- **addrlen** (parametr wyjściowy) - wskaźnik na integer, pod którym zapisany zostanie rozmiar struktury `addr`. Zwykle równy rozmiarowi struktury `sockaddr`.

Funkcja zwraca deskryptor nowego gniazda, które może zostać wykorzystane w funkcjach do przesyłania i odbierania danych za pomocą funkcji `send()` i `recv()` (omówionych dalej). Oryginalne gniazdo wciąż nasłuchuje na nowe połączenia.

W przypadku błędu zwraca wartość -1.

Metody `send()` i `recv()`

Metody `send()` i `recv()` są wykorzystywane do wysyłania oraz odbierania danych w gniazdach połączeniowych, zwracanych przez metodę `accept()` (serwer) lub tych, na których wywołano metodę `connect()` (klient).

Język Python

```

1 socket.send(string[, flags])
2 socket.recv(bufsize[, flags])
3
4 # Przykład:
5 # import socket
6 # s = socket.socket()
7 #
8 # conn, address = s.accept()
9 # req = conn.recv(2048)
10 # conn.send('Hello!')
```

Listing 4.14: Metoda `send()` i `recv()` w Pythonie.

Parametry:

- **string** - dane wysyłane do drugiej strony w połączeniu.
- **flags** - parametr opcjonalny z dodatkowymi flagami, opisanymi w manualu dla `send` lub `recv`.
- **bufsize** - maksymalna liczba bajtów odebrana w ramach wywołania metody.

Metoda `send()` wysyła przez połączenie dane i zwraca liczbę wysłanych bajtów. Metoda `recv()` zwraca odebrane dane, przy czym jeśli metoda zwróci pusty串, to znaczy, że druga strona zamknęła połączenie. Obie wyrzucają wyjątki, gdy wystąpi błąd.

Język C

```

1 #include <sys/socket.h>
2
3 int send(int sockfd, const void *msg, int len, int flags);
4 int recv(int sockfd, void *buf, int len, unsigned int flags);
```

Listing 4.15: Funkcja `send()` i `recv()` w C.

Parametry:

- **sockfd** - deskryptor gniazda do komunikacji.
- **msg** - dane wysyłane przez połączenie.
- **buf** - wskaźnik, pod którym zapisane zostaną odebrane dane.
- **len** - rozmiar (w bajtach) wysyłanych danych przez funkcję `sendto()` lub maksymalna liczba odebranych bajtów przez funkcję `recv()`.

- **flags** - parametr opcjonalny z dodatkowymi flagami, opisanymi w manualu dla `send` lub `recv`. Zwykle ustawiany na 0, co oznacza brak dodatkowych flag.

Funkcja `send()` zwraca liczbę bajtów, które zostały odebrane. Może to być liczba mniejsza niż zadeklarowana w parametrze `len`, ponieważ komputer może nie dać rady wysłać zadeklarowanej liczby bajtów. W takiej sytuacji należy ponownie skorzystać z funkcji `send()`, żeby dosłać pozostałe dane.

Funkcja `recv()` zwraca liczbę odebranych danych. Funkcja oczekuje na odbiór danych dopóki druga strona ich nie wyśle lub zakończy połączenie. W tej drugiej sytuacji funkcja zwraca wartość 0.

W przypadku błędu obie funkcje zwracają wartość -1.

Metody `sendto()` i `recvfrom()`

Odpowiednikami metod `send()` i `recv()` dla gniazd bezpołączeniowych są metody `sendto()` oraz `recvfrom()`. Różnica polega na tym, że w przypadku metod `sendto()` oraz `recvfrom()` w parametrze otrzymują adres i port odbiorcy, ponieważ gniazda nie są połączone z żadnym serwerem.

Gniazdo datagramowe (o typie `SOCK_DGRAM`) również mogą być wykorzystywane przez metodę `connect()` i wtedy można korzystać z metod `send()` i `recv()` do komunikacji, tak jak w przypadku gniazd połączeniowych, mimo że protokołem Warstwy Transportowej pozostaje UDP.

Język Python

```

1 socket.sendto(string, address)
2 socket.sendto(string, flags, address)
3 socket.recvfrom(bufsize[, flags])
4
5 # Przykład:
6 # import socket
7 # s = socket.socket()
8 # ...
9 # conn, address = s.recvfrom(2048)
10 # conn.sendto('Hello!', address)
```

Listing 4.16: Metoda `sendto()` i `recvfrom()` w Pythonie.

Parametry:

- **string** - dane wysyłane do adresata.
- **flags** - parametr opcjonalny z dodatkowymi flagami, opisanymi w manualu dla `sendto` lub `recvfrom`.
- **address** - parametr definiuje adres oraz port, na który dane mają zostać wysłane.

- **bufsize** - maksymalna liczba bajtów odebrana w ramach wywołania metody.

Metoda `sendto()` wysyła na adres i port podany w parametrze `address` dane z parametru `string` i zwraca liczbę wysłanych bajtów. Metoda `recvfrom()` zwraca odebrane dane oraz adres i port, z którego zostały odebrane. Obie wywołują wyjątki, gdy wystąpi błąd.

Język C

```

1 #include <sys/socket.h>
2
3 int sendto(int sockfd, const void *msg, int len, unsigned int flags, const struct
4           sockaddr *to, int tolen);
5 int recvfrom(int sockfd, void *buf, int len, unsigned int flags, struct sockaddr *
               from, int *fromlen);
```

Listing 4.17: Funkcja `sendto()` i `recvfrom()` w C.

Parametry:

- **sockfd** - deskryptor gniazda do komunikacji.
- **msg** - dane wysyłane do adresata.
- **buf** - wskaźnik, pod którym zapisane zostaną odebrane dane.
- **len** - rozmiar (w bajtach) wysyłanych danych przez funkcję `send()` lub maksymalna liczba odebranych bajtów.
- **to** - adres struktury `sockaddr` lub `sockaddr_in`, w której zdefiniowany jest adresat danych.
- **tolen** - rozmiar w bajtach struktury z parametru **to**.
- **from** (parametr wyjściowy) - adres struktury `sockaddr` lub `sockaddr_in`, która zostanie uzupełniona adresem oraz portem komputera, z którego nadeszły dane.
- **fromlen** (parametr wyjściowy) - wskaźnik na integer, pod którym zapisany zostanie rozmiar struktury **to**. Zwykle równy rozmiarowi struktury `sockaddr`.
- **flags** - parametr opcjonalny z dodatkowymi flagami, opisanymi w manualu dla `send` lub `recv`. Zwykle ustawiany na 0, co oznacza brak dodatkowych flag.

Funkcja `sendto()` zwraca liczbę bajtów, które zostały odebrane. Podobnie, jak w przypadku `send()` może to być liczba mniejsza niż zadeklarowana w parametrze `len`, ponieważ komputer może nie dać rady wysłać zadeklarowanej liczby bajtów. W takiej sytuacji należy ponownie skorzystać z funkcji `sendto()`, żeby dosłać pozostałe dane.

Funkcja `recvfrom()` zwraca liczbę odebranych danych. Funkcja oczekuje na odbiór danych dopóki druga strona ich nie wyśle lub zakończy połączenie. W tej drugiej sytuacji funkcja zwraca wartość 0.

W przypadku błędu obie funkcje zwracają wartość -1.

Metody `close()` i `shutdown()`

Po pełnej wymianie komunikacji należy zasyngalizować drugiej stronie chęć jej zakończenia oraz zaprzestać nasłuchiwania na połączenia lub dane. W tym celu można skorzystać z metod `close()` lub `shutdown()`. Pierwsza z nich służy do zamknięcia połączenia i po jej wywołaniu nie można już ani wysyłać, ani odbierać danych przez gniazdo. Druga strona natomiast nie otrzyma już żadnych nowych danych (poza tymi, które zostały wysłane i zakolejkowane). W przypadku połączenia TCP nie jest ono natychmiastowo zrywane, lecz zamykane zgodnie z procesem opisany w sekcji 2.4.2.

Metoda `shutdown()` służy do zamknięcia całego połączenia (podobnie jak `close()`, ale może również być wykorzystana do zamknięcia jednej strony połączenia).

Wywołanie metody `shutdown()` na połączonym gnieździe datagramowym blokuje możliwość wykorzystywania metod `send()` i `recv()`.

Język Python

```

1 socket.close()
2 socket.shutdown(how)
3
4 # Pzykład:
5 # import socket
6 # s = socket.socket()
7 # ...
8 # s.close()

```

Listing 4.18: Metoda `close()` i `shutdown()` w Pythonie.

Parametry:

- **how** - definiuje sposób zamknięcia połączenia. Wartość `SHUT_RD` blokuje dalsze pobieranie danych. Wartość `SHUT_WR` blokuje możliwość wysyłania danych. Natomiast wartość `SHUT_RDWR` blokuje obie czynności.

Metoda wyrzuca wyjątek, gdy wystąpi błąd.

Język C

```

1 #include <sys/socket.h>
2
3 int close(int sockfd);
4 int shutdown(int sockfd, int how);

```

Listing 4.19: Funkcja `close` i `shutdown()` w C.

Parametry:

- **sockfd** - deskryptor gniazda do komunikacji.
- **how** - definiuje sposób zamknięcia połączenia. Wartość SHUT_RD blokuje dalsze pobieranie danych. Wartość SHUT_WR blokuje możliwość wysyłania danych. Natomiast wartość SHUT_RDWR blokuje obie czynności.

Obie funkcje zwracają 0, gdy operacja zostanie zakończona pomyślnie , lub -1 w przypadku błędu.

4.1.2 Metody pomocnicze

Oprócz metod wykorzystywanych do przeprowadzania komunikacji przydatne są również metody dodatkowe, które nie są bezpośrednio związane z transmisją danych.

Metoda `gethostbyname()`

Metoda `gethostbyname()` służy do pobrania adresu IP komputera na podstawie jego nazwy. Czy pamiętasz który protokół odpowiadał za tę operację? Był to DNS, który odpytywał serwer nazw o adres IP domeny.

Język Python

```
1 socket.gethostbyname(hostname)
```

Listing 4.20: Metoda `gethostbyname()` w Pythonie.

Parametry:

- **hostname** - nazwa komputera (np. domena).

Metoda zwraca adres IP komputera w formie napisu, np. '127.0.0.1'. W celu pobrania adresu IP komputera, na którym uruchomiony jest proces można skorzystać z metody `gethostbyname()` przekazując jej jako argument wynik metody `gethostname()`.

Język C

```
1 #include <netdb.h>
2
3 struct hostent *gethostbyname(const char *name);
4
5 struct hostent {
6     char *h_name;
7     char **h_aliases;
8     int h_addrtype;
9     int h_length;
10    char **h_addr_list;
11 };
12 #define h_addr h_addr_list[0]
```

Listing 4.21: Funkcja `gethostbyname()` w C.

Parametry:

- **name** - wskaźnik, pod którym znajduje się nazwa komputera.

Funkcja zwraca strukturę hostent, w której są następujące pola:

- **h_name** - napis zawierający oficjalną nazwę hosta,
- **h_aliases** - tablica napisów zawierających alternatywne nazwy hosta, zakończona wartością NULL,
- **h_addrtype** - typ adresu, czyli wartość AF_INET lub AF_INET6 (omówione wcześniej),
- **h_length** - długość każdego adresu z pola h_addr_list w bajtach,
- **h_addr_list** - tablica wskaźników na adresy IP (w porządku sieciowym) hosta, zakończona wartością NULL.

W przypadku błędu, funkcja zwraca NULL.

Dodatkowo, ze względu na kompatybilność wstępna, definiuje się **h_addr**, która zwraca pierwszy wskaźnik z listy **h_addr_list**.

Metoda gethostname()

Metoda **gethostname()** służy do pobrania nazwy komputera, na którym zostanie uruchomiona.

Język Python

```
1 socket.gethostname()
```

Listing 4.22: Metoda **gethostname()** w Pythonie.

Metoda nie przyjmuje żadnych parametrów i zwraca napis, będący nazwą komputera.

Język C

```
1 #include <unistd.h>
2
3 int gethostname(char *name, size_t len);
```

Listing 4.23: Funkcja **gethostname()** w C.

Parametry:

- **name** (parametr wyjściowy) - wskaźnik, pod którym zostanie wpisana nazwa komputera.
- **len** - parametr określający maksymalną liczbę pobranych bajtów. Jeśli nazwa komputera jest dłuższa niż wartość tego parametru, zostanie obcięta. Innymi słowy jest to rozmiar bufora na nazwę komputera.

Funkcja zwraca liczbę pobranych bajtów, składających się na nazwę komputera. Liczba może być mniejsza niż wartość parametru **len**, gdy nazwa komputera jest krótsza od rozmiaru bufora.

Metoda `getpeername()`

Metoda `getpeername()` służy do pobrania informacji o komputerze, z którym połączone jest gniazdo.

Język Python

```

1 socket.getpeername()
2
3 # Pzykład:
4 # import socket
5 # s = socket.socket()
6 # ...
7 # conn, address = s.accept()
8 # addr = conn.getpeername()

```

Listing 4.24: Metoda `getpeername()` w Pythonie.

Metoda nie przyjmuje żadnych parametrów i zwraca adres zdalny, z którym połączone jest gniazdo, na rzecz którego wywołano metodę. Adres ma taki sam format, jak parametr metody `bind()`, np. ('127.0.0.1', 56443).

Język C

```

1 #include <sys/socket.h>
2
3 int getpeername(int sockfd, struct sockaddr *addr, int *addrlen);

```

Listing 4.25: Funkcja `getpeername()` w C.

Parametry:

- **sockfd** - deskryptor gniazda do komunikacji.
- **addr** (parametr wyjściowy) - adres struktury `sockaddr` lub `sockaddr_in`, która zostanie uzupełniona adresem oraz portem komputera, z którym połączone jest gniazdo.
- **addrlen** (parametr wyjściowy) - adres zmiennej typu `int`, pod którym umieszczono rozmiar w bajtach struktury pod adresem `addr` (rozmiar bufora). Po wywołaniu funkcji, pod tym adresem zapisywany jest faktyczny rozmiar uzupełnionej struktury `addr`. Jeśli faktyczny rozmiar zwracanej struktury się nie mieści w buforze, to jest obcinany, a pod adresem `addrlen` wpisywany jest jego faktyczny rozmiar.

Funkcja zwraca 0, jeśli nie wystąpił błąd lub -1, w przeciwnym razie.

4.1.3 Prosty klient DNS

Pierwszym przykładem programu, w którym skorzystamy z biblioteki `socket` będzie klient DNS, czyli program, który będzie zamieniał nazwę komputera (np. domenę) na jego adres. Nie będziemy tu jeszcze tworzyć gniazda, ale skorzystamy z metody pomocniczej `gethostbyname()`.

Źródła dostępne są w repozytorium w folderze dns_client/.

Język Python

```

1 import sys
2 import socket
3
4 if __name__ == '__main__':
5
6     if len(sys.argv) != 2:
7         sys.stderr.write("usage: dns_client address\n")
8         exit(1)
9
10    hostname = sys.argv[1]
11    ip_address = socket.gethostbyname(hostname)
12    print("IP address: {}".format(ip_address))

```

Listing 4.26: Klient DNS w Pythonie.

Na początku importujemy wymagane biblioteki, czyli `sys` i `socket`.

W linii 4 sprawdzamy, czy skrypt został uruchomiony jako program, czyli np. `python dns_client.py <domain>`. Innym sposobem wykorzystania skryptu jest zimportowanie go, tak jak to zrobiliśmy z bibliotekami (`import dns_client`), jednakże wtedy zmienna `__name__` nie byłaby równa `__main__` i warunek nie zostałby spełniony. Podsumowując, kod z ifa wykona się tylko wtedy, gdy skrypt zostanie uruchomiony jako program, a nie zimportowany jako biblioteka.

W linii 6 sprawdzamy, ile jest elementów w zmiennej `argv` z modulu `sys`. Zmienna ta przechowuje listę parametrów, które zostały przekazane do programu podczas wywołania. Lista ta nie uwzględnia programu `python`, ale uwzględnia nazwę naszego skryptu `dns_client`. Do dalszego działania wymagane są 2 parametry, czyli nazwa naszego skryptu (dodawana do listy automatycznie) oraz jeszcze jeden parametr, czyli sprawdzana domena. Jeśli liczba parametrów będzie różna od 2, to wyświetlimy na standardowym wyjściu (`sys.stderr.write`) komunikat z informacją, w jaki sposób należy wywoływać program i kończymy działanie funkcją `exit` z wartością 1, oznaczającą kod błędu.

W linii 10 zapisujemy w zmiennej `hostname` wartość parametru z domeną, by przekazać go do funkcji `gethostbyname` z modulu `socket`. Wynikiem funkcji jest adres IP, który wyświetlamy w linii 12.

Język C

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <errno.h>
4 #include <netdb.h>
5 #include <arpa/inet.h>
6
7 int main(int argc, char *argv[])
8 {
9     struct hostent *h;

```

```

10 |     if (argc != 2) {
11 |         fprintf(stderr, "usage: dns_client address\n");
12 |         exit(1);
13 |     }
14 |
15 |     if ((h=gethostbyname(argv[1])) == NULL) { // pobierz informacje o hoscie
16 |         perror("gethostbyname");
17 |         exit(1);
18 |     }
19 |     printf("IP Address : %s\n", inet_ntoa(*((struct in_addr *)h->h_addr)));
20 |     return 0;
21 |

```

Listing 4.27: Klient DNS w C.

Kompilacja źródeł: gcc -o dns_client dns_client.c
Uruchomienie programu: ./dns_client

Na początku dołączamy wszystkie wymagane pliki nagłówkowe.

W linii 7 rozpoczynamy funkcję main, która reprezentuje główną funkcję uruchamianą jako pierwszą podczas uruchamiania skompilowanego programu. Funkcja przyjmuje 2 parametry: pierwszy z nich argc przechowuje liczbę przekazanych argumentów do programu (włączając jego nazwę), zaś drugi argv to tablica napisów, gdzie każdy z nich to parametr przekazany do programu, przy czym pierwszym parametrem jest zawsze nazwa programu (tutaj dns_client).

W linii 9 deklarujemy zmienną wskaźnikową h na strukturę hostent.

W linii 10 sprawdzamy, czy liczba argumentów jest równa 2, czyli czy przekazano jedną nazwę domeny. Jeśli liczba argumentów będzie różna od 2, to wyświetlamy na standardowym wyjściu (fprintf(stderr, "...")) komunikat z informacją, w jaki sposób należy wywoływać program i kończymy działanie funkcją exit z wartością 1, oznaczającą kod błędu.

W linii 15 mamy ifa, w którym sprawdzamy wynik funkcji gethostbyname przypisany do zmiennej h. Funkcja ta przyjmuje drugim argumentem programu jako parametr, czyli przekazaną nazwę. Sprawdzenie wyniku polega na porównaniu go do wartości NULL, która jest zwracana przez funkcję, gdy wystąpił błąd. Jeśli funkcjawróciła NULL to wyświetlamy komunikat błędu za pomocą funkcji perror oraz kończymy działanie funkcją exit z wartością 1, oznaczającą kod błędu.

Funkcje w C same nie zwracają błędu, lecz ustawiają globalną zmienną errno na wartość odpowiadającą występującemu błędowi, a zwracają wartość ogólnie oznaczającą błąd, np. -1 lub NULL. W celu wyświetlenia informacji o błędzie należy zinterpretować samemu wartość errno lub skorzystać z funkcji perror(char *), która wyświetla na standardowym wyjściu błędu przekazany parametr oraz zinterpretowaną informację o błędzie np. perror("fopen") wyświetli fopen: No such file or directory dla wartości errno odpowiadającej wartości mówiącej o tym, że plik, do którego chcemy się dostać nie istnieje.

W powyższym przykładzie nie wykorzystujemy funkcji perror, tylko perror, ponieważ funkcje sieciowe (np. gethostbyname) nie ustawiają błędu w zmiennej errno, tylko w zmiennej h_errno, do której odwołuje się funkcja h_error. Poza tą różnicą perror działa tak samo, jak perror.

Jeśli wartość zwrócona przez funkcję gethostbyname i zapisana w zmiennej jest różna od NULL, to przechodzimy do linii 19. W niej wyświetlamy napis IP Address, a za nim rzutujemy zmienną h na wskaźnik na strukturę in_addr, żeby dobrać się do pola h_addr (tak naprawdę nie ma takiego pola, tylko to jest alias dobierania się do pierwszego elementu pola h_addr_list, patrz 4.21). Następnie wartość tego pola jest zamieniana na napis za pomocą funkcji inet_ntoa.

4.1.4 Prosta aplikacja strumieniowa

W drugim przykładzie wykorzystamy już gniazda i stworzymy połączenie, czyli wykorzystamy protokół TCP. Będzie to przykład, w którym napiszemy server Echo, czyli taki, który odpowiada klientowi tym samym, co otrzymał. Napiszemy również klienta dla tego serwera.

Źródła dostępne są w repozytorium w folderze `tcp_example/`.

Język Python

```

1 import sys
2 import socket
3
4 BUF_SIZE = 1024
5
6 if __name__ == "__main__":
7
8     if len(sys.argv) != 2:
9         sys.stderr.write("usage: tcp_server port\n")
10        exit(1)
11
12    try:
13        port = int(sys.argv[1])
14        assert port > 0
15    except:
16        sys.stderr.write("error: invalid port\n")
17        exit(1)
18
19    sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
20    sock.bind(("0.0.0.0", port))
21    sock.listen(5)
22
23    while True:
24        client, addr = sock.accept()
25        print(addr[0], "connected now.")
26
27        while True:
28            data = client.recv(BUF_SIZE)
29            if not data:
30                break

```

```

31     client.send(data)
32
33     client.close()
34
35     sock.close()

```

Listing 4.28: Serwer ECHO w Pythonie

Podobnie jak w przykładzie poprzednim najpierw sprawdzamy, czy skrypt został wywołany z poprawnym argumentem (linie 8-17). W linii 13 rzutujemy przekazany argument na liczbę całkowitą i operację tę umieszczamy w bloku `try ... except`, ponieważ jeśli przekazany napis nie jest liczbą całkowitą, to funkcja `int` wyrzuci wyjątek. W linii poniżej sprawdzam, czy port jest większy od zera, poprzedzając warunek słowem kluczowym `assert`. Jego działanie polega na tym, że oczekuje na warunek zwracający prawdę, a jeśli się tak nie stanie, to wyrzuca wyjątek `AssertionError`, który również zostanie przechwycony przez sekcję `except`.

Po sprawdzeniu poprawności argumentów tworzę w linii 19 gniazdo strumieniowe, po czym wiążę je z portem przekazanym w argumencie i adresem `0.0.0.0`. Adres ten oznacza, że akceptuje połączenia z dowolnego adresu IP (oczywiście pod warunkiem, że firewall nie blokuje połączeń). Jeśli chcielibyśmy dać możliwość łączenia się z naszym serwerem tylko z tego samego komputera, to moglibyśmy umieścić adres `127.0.0.1`. W linii 21 rozpoczynam nasłuchiwanie na połączenia i zaznaczam, że w kolejce może być maksymalnie 5 oczekujących połączeń.

W linii 23 wykorzystujemy nieskończoną pętlę, ponieważ nasz serwer raz uruchomiony ma odpowiadać na kolejne próby połączeń, a my nie wiemy ile takich połączeń będzie. Serwery działają w tle i zwykle kończą działanie, gdy są manualnie wyłączane przez administratora.

W linii 24 czekamy na nadchodzące połączenie i gdy się ono pojawi, metoda zwraca gniazdo tego połączenia `client` oraz adres, z którego połączenie nadeszło `addr`. W następnej linii wykorzystujemy pierwszy element zwróconego adresu, czyli adres IP, który wyświetlamy na ekranie.

Dopiero w linii 27 zaczyna się właściwa logika naszego serwera. Wcześniejsze instrukcje będą się powtarzały w większości serwerów. W linii 27 uruchamiamy drugą pętlę nieskończoną, ponieważ nasz serwer ma przyjmować wiadomość od nadawcy i odsyłać ją, natomiast nie wiemy ile tych wiadomości nadawca będzie chciał wysłać.

W linii 28 pobieramy wiadomość od nadawcy. Jej maksymalna długość jest określona przez zmienną `BUFSIZE`. Sprawdź co się dzieje, gdy ustawisz niewielką wartość tej zmiennej (np. 2). Po otrzymaniu wiadomości sprawdzamy, czy nie jest ona pusta, ponieważ taka sytuacja oznaczałaby, że nadawca zakończył połączenie. Jeśli wiadomość jest pusta to wychodzimy w wewnętrznej pętli. Jeśli natomiast wiadomość `data` nie jest pusta, to odsyłamy ją w linii 31.

Po wyjściu z wewnętrznej pętli zamknięte jest połączenie z klientem (linia 33). W skrypcie jest również zamknięcie gniazda głównego w linii 35, jednakże skrypt do tej linii nigdy nie dotrze, ponieważ nie ma możliwości, żeby opuścił pętlę z linii 23. Operacja ta jest dodana dla formalności, żeby o niej pamiętać, bo

tak naprawdę w prawdziwym skrypcie serwera powinna być obsłużona sytuacja, w której opuszczamy główną pętlę.

Zakończenie działania takiego serwera to po prostu skorzystanie ze skrótu Ctrl+C.

Język C

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <errno.h>
4 #include <sys/types.h>
5 #include <sys/socket.h>
6 #include <netinet/in.h>
7 #include <arpa/inet.h>
8 #include <unistd.h>
9
10 int main(int argc, char * argv[]) {
11     int BUF_SIZE = 1024;
12     int port;
13     int sockfd, connectedfd;
14     struct sockaddr_in listen_addr;
15     struct sockaddr_in client_addr;
16     int sin_size, recv_size;
17     char buffer[BUF_SIZE+1];
18
19     sin_size = sizeof(struct sockaddr_in);
20     if (argc != 2) {
21         fprintf(stderr,"usage: tcp_server port\n");
22         exit(1);
23     }
24     port = atoi(argv[1]);
25     if (port <= 0) {
26         fprintf(stderr,"error: invalid port\n");
27         exit(1);
28     }
29     if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
30         perror("socket");
31         exit(1);
32     }
33
34     listen_addr.sin_family = AF_INET;
35     listen_addr.sin_port = htons(port);
36     listen_addr.sin_addr.s_addr = INADDR_ANY;
37
38     if (bind(sockfd, (struct sockaddr *)&listen_addr, sizeof(struct sockaddr)) ==
39         -1) {
40         perror("bind");
41         exit(1);
42     }
43     if (listen(sockfd, 5) == -1) {
44         perror("listen");
45         exit(1);
46     }
47     while(1) {
48         if ((connectedfd = accept(sockfd, (struct sockaddr *)&client_addr, &sin_size)) ==
49             -1) {
50             perror("accept");
51             continue;
52         }
53         printf("%s connected now.\n", inet_ntoa(client_addr.sin_addr));
54         while(1) {
55             if ((recv_size = recv(connectedfd, buffer, BUF_SIZE, 0)) == 0) {
56                 close(connectedfd);
57                 if (errno != 0) {
58                     perror("recv");
59                 }
60             }
61         }
62     }
63 }
```

```

58     break;
59 }
60 buffer[recv_size] = '\0';
61 if (send(connect(sockfd, buffer, recv_size, 0) == -1) {
62     perror("send");
63     close(sockfd);
64     break;
65 }
66 }
67 }
68 }
```

Listing 4.29: Serwer ECHO w C

Na początku funkcji `main` deklarujemy wszystkie zmienne (zgodnie z ANSI-C), które będę dokładniej opisywał w dalszej części kodu.

W linii 20 sprawdzamy liczbę argumentów (muszą być dwa, czyli nazwa programu i port), zaś w linii 24 zamieniamy napis na liczbę całkowitą i sprawdzamy wynik. Jeżeli podczas uruchomienia podano napis, który nie jest liczbą to funkcja zwraca wartość 0, dlatego poniżej sprawdzamy, czy wartość `port` jest mniejsza od 0 i jeśli jest, to kończymy program w komunikatem o błędzie.

W linii 29 tworzymy gniazdo strumieniowe i jego deskryptor zapisujemy w `sockfd`, a także od razu sprawdzamy, czy nie jest równe -1, co by znaczyło, że gniazda nie udało się stworzyć. Za każdym razem, gdy dowolna funkcja nie wykona się pomyślnie, czyli np. zwróci -1, przerywamy działanie programu.

W liniach 34-36 ustalamy adres nasłuchiwanego, przypisując port podany w argumencie i zamieniony na format sieciowy oraz adres równy `INADDR_ANY`, oznaczający, że nasłuchujemy na połączenia z dowolnego adresu IP (oczywiście zakładając, że firewall połączenie przepuści).

W linii 38 wiążemy gniazdo z przygotowanym adresem nasłuchiwanego i podobnie jak w przypadku funkcji `socket` sprawdzamy, czy funkcja zwróciła wartość -1, czyli wystąpił błąd.

Linia 46 to pętla nieskończona, ponieważ nasz serwer raz uruchomiony ma odpowiadać na kolejne próby połączeń, a my nie wiemy ile takich połączeń będzie. Serwery działają w tle i zwykle kończą działanie, gdy są manualnie wyłączane przez administratora.

W linii 47 czekamy na nadchodzące połączenie i gdy się ono pojawi, funkcja zwraca deskryptor gniazda tego połączenia `connectedfd` oraz zapisuje w zmiennej `client_addr` adres, z którego nadeszło połączenie. Tutaj również sprawdzamy, czy funkcja nie zakończyła działania z błędem. W linii 51 wyświetlamy na ekranie adres IP, z którego nadeszło połączenie, wykorzystując pole `sin_addr` ze zmiennej `client_addr`. Wcześniej oczywiście zamieniamy go z sieciowego formatu liczbowego na napis.

Dopiero w linii 52 zaczyna się właściwa logika naszego serwera. Wcześniejsze instrukcje będą się powtarzały w większości serwerów. W linii 52 uruchamiamy drugą pętlę nieskończoną, ponieważ nasz serwer ma przyjmować wiadomość od nadawcy i odsyłać ją, natomiast nie wiemy ile tych wiadomości nadawca będzie chciał wysłać.

W linii 53 pobieramy wiadomość od nadawcy. Jej maksymalna długość jest określona przez zmienną `BUFSIZE`, sama wiadomość zapisywana jest w

zmiennej `buffer`, a jej długość zostanie zwrócona przez funkcję i zapisana w `recv_size`. Sprawdź co się dzieje, gdy ustawisz niewielką wartość zmiennej `BUF_SIZE` (np. 2). Po otrzymaniu wiadomości sprawdzamy, czy nie jest ona pusta (czyli wartość `recv_size` jest równa 0), ponieważ taka sytuacja oznaczałaby, że nadawca zakończył połączenie lub wystąpił błąd. Jeśli `recv_size` jest równe 0 to zamykamy deskryptor połączenia i wychodzimy wewnętrznej pętli. Wcześniej jednak sprawdzamy wartość `errno`, która równa 0 oznacza, że druga strona po prostu zamknęła połączenie (nie ma żadnego błędu), jednakże inna wartość oznacza, że wystąpił jakiś błąd, dlatego wyświetlamy jego treść za pomocą funkcji `perror`.

Jeśli natomiast `recv_size` jest większe od 0, to dodajemy za ostatnim pobranym znakiem znak

0, oznaczający koniec napisu (linia 60). Dodanie znaku jest potrzebne w sytuacji, gdy chcielibyśmy zmienną `buffer` wykorzystać jako napis, np. wyświetlając ją. Następnie za pomocą funkcji `send` odsyłamy wartość zmiennej `buffer` o takiej samej długości, jaką otrzymaliśmy, czyli `recv_size`. Tutaj ponownie sprawdzamy, czy funkcja nie zwróciła wartości -1 oznaczającej błąd.

W skrypcie nie ma zamknięcia deskryptora gniazda głównego poza zewnętrzną pętlą, ponieważ skrypt do tej linii nigdy by nie dotarł. Po prostu nie ma możliwości, żeby opuścił pętlę z linii 46. Tak naprawdę w prawdziwym skrypcie serwera powinna być obsłużona sytuacja, w której opuszczamy główną pętlę.

Zakończenie działania takiego serwera to po prostu skorzystanie ze skrótu `Ctrl+C`.

Klient TCP

Mamy gotowy serwer, teraz pora zająć się klientem. Nie do końca, bo tak naprawdę mamy klienta pod ręką. Jest nim znany program `telnet` lub również popularny linuksowy `netcat`. W przykładzie i dalej w skrypcie będę korzystał z programu `netcat`, opisywanego jako *Swiss army knife for network connections*.

Istnieją dwie implementacje programu `netcat`, które delikatnie różnią się flagami i funkcjonalnościami, jednakże w ramach tych zajęć nie są one istotne, także możesz używać komendy `nc` bez względu na to, która implementacja się pod nią kryje.

Jak uruchomimy nasz serwer (nieważne czy w Pythonie, czy w C po skompilowaniu) np. z portem 9999 to w drugim terminalu możemy się z nim połączyć za pomocą komendy `nc <adresIP> 9999`, gdzie zamiast pola `<adresIP>` wpisujemy adres komputera, na którym uruchomiliśmy serwer. Jeśli łączymy się z tego samego komputera, to adres jest równy `127.0.0.1`, czyli `localhost`.

Na Rysunku 4.2 umieściłem zrzut ekranu z dwóch terminali. Po lewej stronie uruchomiłem serwer (skrypt Python), zaś po prawej połączylem się z nim z tego samego komputera (a dokładniej kontenera) i wysłałem dwie wiadomości TEST

The screenshot shows two terminal windows side-by-side. The left window, titled 'root@815fb72aa16e:/opt/pas', contains Python code for a TCP server. It starts with imports for sys and socket, defines a BUF_SIZE of 1024, and checks if the script is run as the main program. It then handles command-line arguments for address and port, connects to the specified host and port, and enters a loop to receive and send data. The right window, also titled 'root@815fb72aa16e:/opt/pas', shows the netcat client connected to the server at port 9999. It sends four lines of text: 'TEST', 'TEST', 'TEST2', and 'TEST2'. The server's response is visible in the left window's terminal.

```

root@815fb72aa16e:/opt/pas# python tcp_example/tcp_server.py 9999
('127.0.0.1', 'connected now.')
^C[Traceback (most recent call last):
  File "tcp_example/tcp_server.py", line 28, in <module>
    data = client.recv(BUF_SIZE)
KeyboardInterrupt
root@815fb72aa16e:/opt/pas# nc 127.0.0.1 9999
TEST
TEST
TEST2
TEST2
root@815fb72aa16e:/opt/pas#

```

Rysunek 4.2: Netcat - połączenie z serwerem TCP.

oraz TEST2. Jak widać netcat od razy wyświetlił odpowiedź odeslaną przez serwer, czyli drugi raz tę samą wiadomość.

Mamy zatem gotowego klienta, więc czy jest sens pisać go samemu? Niby nie, jednakże ten skrypt ma za zadanie nauczyć programowania aplikacji sieciowych, a nieodłączną częścią serwera jest klient, dlatego poniżej prezentuje prostego klienta dla naszego serwera. Tak naprawdę będzie to mocno okrojony netcat.

W przypadku wykorzystania serwera obsługującego konkretny protokół (np. SMTP) nie obejdziemy się oczywiście bez klienta dedykowanego dla tego protokołu. Inaczej pocztę przeglądaliśmy za pomocą netcata, a nie np. Outlooka. Nie oznacza to, że za pomocą netcata nie wyśle lub przeczytam maila, ale w takiej sytuacji musiałbym znać na pamięć protokoły i wysyłać netcatem odpowiednie komendy.

Język Python

```

1 import sys
2 import socket
3
4 BUF_SIZE = 1024
5
6 if __name__ == "__main__":
7
8     if len(sys.argv) != 3:
9         sys.stderr.write("usage: tcp_client ip port\n")
10        exit(1)
11
12     try:
13         addr = sys.argv[1]
14         port = int(sys.argv[2])
15         assert port > 0
16     except:
17         sys.stderr.write("error: invalid port\n")
18         exit(1)
19
20     sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
21     try:
22         sock.connect((addr, port))
23
24         while True:
25             data = raw_input('Data to send (empty line quits):')
26             if data == '':
27                 break
28             sock.send(data)
29             data = sock.recv(BUF_SIZE)
30             if not data:
31                 break

```

```

32     print 'Answer:', data
33     sock.close()
34
35 except socket.error, e:
36     print 'Error:', e

```

Listing 4.30: Klient ECHO w Pythonie

Na początku, w liniach 8 - 18, weryfikowana jest poprawność argumentów. Różnica między klientem, a serwerem polega na tym, że klient posiada jeszcze jeden argument, czyli adres serwera, z którym chce się połączyć, zapisywany w zmiennej `addr`.

W linii 20 tworzymy nowe gniazdo, tak samo jak w serwerze.

Dalszą część kodu umieściłem w bloku `try ... except`, żeby ewentualne błędy połączenia i inne wyrzucające wyjątek `socket.error` były wyświetlane na ekranie.

W linii 22 łączymy gniazdo z serwerem przekazując krotkę z adresem oraz portem serwera. Następnie uruchamiamy nieskończoną pętlę, w której będziemy użytkownika pytać o dane do wysłania, wysyłać je oraz wyświetlać odpowiedź.

Dane od użytkownika pobieramy za pomocą funkcji `raw_input`, która wyświetla prompt *Data to send (empty line quits)*. Jeśli użytkownik nie poda żadnej treści, czyli od razu wcisnie Enter, to warunek z linii 26 zostanie spełniony i program opuści pętlę. Jeśli użytkownik podał dane, to zostaną one wysłane do serwera w linii 28 za pomocą metody `send`.

Następnym krokiem jest odebranie odpowiedzi z serwera w linii 29, która jest zapisywana w zmiennej `data`. Metoda `recv` otrzymuje jeden argument równy zmiennej `BUF_SIZE`, który określa maksymalny rozmiar odebranej odpowiedzi. Jeśli odpowiedź serwera byłaby większa niż maksymalny rozmiar, to należy wywołać metodę `recv` ponownie, aby odebrać kolejną porcję odpowiedzi. Zwykle w klientach docelowych przy każdej wiadomości odbiera się dane, dopóki nie odbierze się konkretnej wartości oznaczającej koniec wiadomości. Wspomnimy jeszcze o tym później.

Warunek w linii 30 służy do sprawdzenia, czy serwer nie zamknął połączenia. Analogiczny warunek umieściliśmy wcześniej w kodzie serwera. Na końcu pętli wyświetlamy otrzymaną odpowiedź, natomiast za pętlką zamykamy połączenie za pomocą metody `close`. Tutaj, w przeciwnieństwie do serwera, klient dotrze, bo wystarczy, że użytkownik poda pusty napis do wysłania.

Język C

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4 #include <errno.h>
5 #include <sys/types.h>
6 #include <sys/socket.h>
7 #include <netinet/in.h>
8 #include <arpa/inet.h>
9 #include <unistd.h>
10
11 int main(int argc, char * argv[]) {
12     int BUF_SIZE = 1024;
13     char * ip_addr;

```

```

14 int port;
15 int sockfd;
16 struct sockaddr_in server_addr;
17 int recv_size, read_size, sent_size;
18 char buffer[BUF_SIZE+1];
19
20 if (argc != 3) {
21     fprintf(stderr,"usage: tcp_client ip port\n");
22     exit(1);
23 }
24 ip_addr = argv[1];
25 port = atoi(argv[2]);
26 if (port <= 0) {
27     fprintf(stderr,"error: invalid port\n");
28     exit(1);
29 }
30 if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
31     perror("socket");
32     exit(1);
33 }
34 server_addr.sin_family = AF_INET;
35 server_addr.sin_port = htons(port);
36 inet_aton(ip_addr, &server_addr.sin_addr);
37
38 if (connect(sockfd, (struct sockaddr *)&server_addr, sizeof(struct sockaddr)) ==
39     -1) {
40     perror("connect");
41     exit(1);
42 }
43 while(1) {
44     printf("Data to send (Ctrl+D quits): ");
45     if (fgets(buffer, BUF_SIZE, stdin) == NULL) {
46         close(sockfd);
47         if (errno != 0) {
48             perror("fgets");
49         }
50         exit(1);
51     }
52     read_size = strlen(buffer);
53
54     if ((sent_size = send(sockfd, buffer, read_size, 0)) == -1) {
55         close(sockfd);
56         perror("send");
57         exit(1);
58     }
59
60     if ((recv_size = recv(sockfd, buffer, BUF_SIZE, 0)) == 0) {
61         close(sockfd);
62         if (errno != 0) {
63             perror("recv");
64             exit(1);
65         }
66         break;
67     }
68
69     buffer[recv_size] = '\0';
70     printf("Answer: %s", buffer);
71 }
72 }
```

Listing 4.31: Klient ECHO w C

Na początku funkcji main deklarujemy wszystkie zmienne (zgodnie z ANSI-C), które będę dokładniej opisywał w dalszej części kodu.

W liniach 20-29, podobnie jak w serwerze, sprawdzamy poprawność argu-

mentów, z tą różnicą, że teraz muszą być trzy: nazwa programu, adres serwera oraz port serwera. Adres zapisujemy w zmiennej `ip_addr`, zaś port w `port`.

Następnie w linii 30, również tak samo jak w serwerze, tworzymy gniazdo i zapisujemy jego deskryptor w polu `sockfd`. Oczywiście sprawdzamy, czy nie wystąpił błąd i kończymy program w takiej sytuacji.

Następnie przygotowujemy zmienną `server_addr`, która będzie zawierała adres serwera, z którym chcemy się połączyć. Ustawiamy pole `sin_family` na `AF_INET`, czyli rodzinę protokołów IPv4, pole `sin_port` na podany port, zamieniony na format sieciowy, dzięki funkcji `htonl` oraz pole `sin_addr` ustawiamy na wartość adresu podanego w argumencie, zamienionego na format sieciowy za pomocą funkcji `inet_aton`.

Kolejny krok to połączenie gniazda z serwerem za pomocą funkcji `connect` w linii 39. Do funkcji przekazujemy wcześniej ustalony deskryptor `sockfd` i przygotowany adres `server_addr`, a także rozmiar struktury przechowującej adres serwera. Jeśli nie wystąpił błąd, to idziemy dalej.

Uruchamiamy pętle nieskończoną, w której będziemy użytkownika pytać o dane do wysłania, wysyłać je oraz wyświetlać odpowiedź.

Pierwszy krok to wyświetlenie za pomocą funkcji `printf` użytkownikowi prośby o podanie danych. Następnie w linii 45 pobieramy za pomocą funkcji `fgets` napis od użytkownika do zmiennej `buffer` o maksymalnej długości `BUF_SIZE`. Funkcja `fgets` pobierze całą linię od użytkownika, czyli cały napis do znaku nowej linii, wliczając znak nowej linii. Sprawdzamy, czy `fgets` nie zwróciło `NULL`. Jeżeli tak to znaczy, że albo wystąpił błąd i wtedy wyświetlamy jego treść i kończymy program albo użytkownik wcisnął `Ctrl+D` i wtedy wartość zmiennej `errno` będzie równa 0, czyli nie wyświetlamy błędu (którego treść w takim przypadku brzmi `Success`), tylko wychodzimy z pętli.

Po wczytaniu danych od użytkownika, zapisujemy ich długość w zmiennej `read_size` (linia 52) i próbujemy je wysłać w linii 54. Do funkcji `send` przekazujemy deskryptor gniazda, dane `buffer` oraz liczbę wysyłanych bajtów, czyli wartość `read_size`, bo chcemy wysłać wszystko, co otrzymaliśmy od użytkownika. Jeśli wystąpił błąd, to go wyświetlamy i kończymy program z kodem 1, a jeśli nie to idziemy dalej.

Po wysłaniu danych oczekujemy na odpowiedź w linii 60. Funkcji `recv` przekazujemy deskryptor gniazda (jak we wszystkich funkcjach dotyczących gniazda), adres `buffer`, pod którym mają być zapisane pobrane dane oraz rozmiar bufora `BUF_SIZE`, a tak naprawdę liczbę o jeden mniejszą od rozmiaru (patrz linia 17), żeby było miejsce na wstawienie `NULL` na koniec napisu. Ostatni argument to 0, czyli brak dodatkowych flag.

W zmiennej `recv_size` zapisana zostanie liczba faktycznie pobranych bajtów. Jeśli będzie ona równa 0 to znaczy, że albo wystąpił błąd albo serwer zakończył połączenie. W obu przypadkach zamykamy połączenie ze strony klienta. Natomiast jeśli wystąpił błąd to `errno` jest większe od zera, więc wyświetlimy w linii 63 błąd oraz zakończymy działanie programu z kodem 1. Natomiast jeśli serwer zakończył połączenie to po prostu wyjdziemy z naszej nieskończonej pętli.

Po poprawnym pobraniu odpowiedzi, w linii 69 dodajemy na końcu popra-

nych danych znak końca napisu (NULL) i wyświetlamy go na ekranie za pomocą funkcji `printf` w linii 70.

Powysze serwery oraz klienci są prostymi aplikacjami, które mają wiele ograniczeń. Na przykład nie pobierają pełnych wiadomości, gdy ich rozmiar jest większy od rozmiaru bufora. Nie są również w stanie obsłużyć kilka połączeń jednocześnie, czy nie mają pełnej obsługi błędów, a jak mają to obsługa polega na zakończeniu działania. Także zakładają znany z góry przepływ informacji, czyli np. najpierw wysyła coś klient, a później odpowiada serwer itp., jednakże często jest taka sytuacja, że w dowolnej chwili każda ze stron może coś wysłać (np. chat).

Wszystkie te ograniczenia będziemy poruszać w kolejnych częściach tak, żeby na koniec zbudować pełnoprawny serwer i klienta.

4.1.5 Prosta aplikacja bezpołączeniowa (UDP)

W poprzednim podrozdziale napisaliśmy parę serwera i klienta wykorzystującego połączenie TCP. W tym natomiast skorzystamy z bezpołączeniowego protokołu UDP.

Różnic będzie niewiele i tylko te różnice opiszę poniżej, ponieważ pozostała część kodu będzie miała takie samo wyjaśnienie, jak w przypadku serwera i klienta TCP.

Największa różnica to różnica projektowa. Teraz nie możemy myśleć w kategorii połączeń, teraz nie ma połączenia między dwoma stronami, po czym wymieniamy pakiety. Teraz otrzymujemy pakiet *skądś* i wysyłamy dokłads, nie jesteśmy ograniczeni połączeniem.

Akurat w przykładzie założenie jest analogiczne, czyli nawiązujemy połączenie do wymiany danych między dwiema stronami, jednakże jest to założenie projektowe, a nie techniczne - wynikające z protokołu.

Język Python

```
1 import sys
2 import socket
3
4 BUF_SIZE = 1024
5
6 if __name__ == "__main__":
7
8     if len(sys.argv) != 2:
9         sys.stderr.write("usage: udp_server port\n")
10        exit(1)
11
12     try:
13         port = int(sys.argv[1])
14         assert port > 0
15     except:
16         sys.stderr.write("error: invalid port\n")
17         exit(1)
18
```

```

19|     sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
20|     sock.bind(("0.0.0.0", port))
21|
22|     while True:
23|         data, addr = sock.recvfrom(BUF_SIZE)
24|         print(addr[0], "sent a message.")
25|
26|         sock.sendto(data, addr)
27|
28|     sock.close()

```

Listing 4.32: Serwer ECHO w Pythonie

Pierwsza zmiana to zmiana napisu w linii 9. Teraz nasz program to `udp_server`, a nie `tcp_server`. Właściwie, to moglibyśmy skorzystać z wartości `sys.argv[0]`, która przechowuje faktyczną nazwę naszego skryptu, bo przecież nie możemy zbroić nikomu zmiany nazwy pliku. Dalsze zmiany mają już wymiar techniczny.

W linii 19 tworzymy gniazdo datagramowe, a nie strumieniowe, czyli jako drugi argument podajemy `SOCK_DGRAM`.

Dalej nie używamy metod `listen` i `accept`, ponieważ nie będziemy oczekiwali na połączenia, a na same pakiety. Stąd, nie mamy dwóch pętli nieskończonych (jedna na połączenia, a druga na pakiety z danego połączenia), tylko jedną pętlę nieskończoną w linii 22 na wszystkie pakiety.

W gniazdach strumieniowych metoda `accept` przymywała połączenie, a mówiąc bardziej technicznie, przeprowadzała z klientem procedurę *3-Way Handshake*. W gniazdach datagramowych tego nie ma.

Oczywiście musimy wiedzieć skąd pakiety przyszły, a skoro nie ma `accept`, to inna metoda musi nam o tym mówić. W linii 23 używamy metody `recvfrom` (zamiast `recv`), która oprócz danych `data` zwraca jeszcze jedną wartość, czyli adres, z którego przyszedł pakiet.

Nie tworzymy nowego połączenia, więc nie jest tworzone żadne nowe gniazdo, a odpowiedź wysyłamy za pomocą tego samego gniazda. W linii 26 korzystamy z metody `sendto`, której pierwszym parametrem są dane, zaś drugim jest adres, na który ma zostać wysłany pakiet. W naszym przykładzie wykorzystujemy adres, z którego przyszedł poprzedni pakiet (linia 23), stąd możemy nowy pakiet nazwać odpowiedzią.

```

1 import sys
2 import socket
3
4 BUF_SIZE = 1024
5
6 if __name__ == "__main__":
7
8     if len(sys.argv) != 3:
9         sys.stderr.write("usage: udp_client ip port\n")
10        exit(1)
11
12     try:
13         addr = sys.argv[1]
14         port = int(sys.argv[2])
15         assert port > 0
16     except:
17         sys.stderr.write("error: invalid port\n")
18         exit(1)
19
20     sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

```

```

21  try:
22      while True:
23          data = raw_input('Data to send (empty line quits):')
24          if data == '':
25              break
26          sock.sendto(data, (addr, port))
27          data, recv_addr = sock.recvfrom(BUF_SIZE)
28          print 'Answer:', data
29          sock.close()
30
31 except socket.error, e:
32     print 'Error:', e

```

Listing 4.33: Klient ECHO w Pythonie

W kliencie również zmieniamy nazwę programu na `udp_client`.

Podobnie, jak w przypadku serwera tworzymy gniazdo datagramowe, a nie strumieniowe, czyli w linii 20 drugi argument zmieniamy na `SOCK_DGRAM`.

Jak wspomniałem w przypadku serwera, gniazdo datagramowe nie nawiązuje połączenia, dlatego nie używamy metody `connect`. W zamian, po wczytaniu danych od użytkownika, przekazujemy do metody `sendto` (linia 26) zarówno dane do wysłania, jak również adres docelowy (wcześniej był on wykorzystywany w metodzie `connect`). Oczywiście adres ten może się zmieniać, podczas gdy w przypadku gniazd strumieniowych wszystkie dane wysyłane za pomocą jednego strumienia miały tego samego adresata.

Następnie w linii 27 pobieramy dane z *jakiegoś* adresu `recv_addr` i w linii 28 wyświetlamy je na ekranie. Celowo dodałem słowo *jakiegoś*, ponieważ nie mamy pewności, że pobrane dane pochodzą z tego samego adresu, na który wcześniej wysłaliśmy dane (w gniazdach strumieniowych mamy taką pewność). Stąd, jeśli chciałbym mieć pewność, że to pakiet z odpowiedzią to powiniensem porównać adres docelowy z linii 26 (zmienne `addr` i `port`) z adresem nadawcy z linii 27 (zmienna `recv_addr`, która zawiera adres IP oraz port, z którego wysłano pakiet).

Język C

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <errno.h>
4 #include <sys/types.h>
5 #include <sys/socket.h>
6 #include <netinet/in.h>
7 #include <arpa/inet.h>
8 #include <unistd.h>
9
10 int main(int argc, char * argv[]) {
11     int BUF_SIZE = 1024;
12     int port;
13     int sockfd;
14     struct sockaddr_in listen_addr;
15     struct sockaddr_in client_addr;
16     int sin_size, recv_size;
17     char buffer[BUF_SIZE+1];
18
19     if (argc != 2) {
20         fprintf(stderr,"usage: udp_server port\n");
21         exit(1);
22     }

```

```

23|     port = atoi(argv[1]);
24|     if (port <= 0) {
25|         fprintf(stderr, "error: invalid port\n");
26|         exit(1);
27|     }
28|     if ((sockfd = socket(AF_INET, SOCK_DGRAM, 0)) == -1) {
29|         perror("socket");
30|         exit(1);
31|     }
32|
33|     listen_addr.sin_family = AF_INET;
34|     listen_addr.sin_port = htons(port);
35|     listen_addr.sin_addr.s_addr = INADDR_ANY;
36|
37|     if (bind(sockfd, (struct sockaddr *)&listen_addr, sizeof(struct sockaddr)) ==
38|         -1) {
39|         perror("bind");
40|         exit(1);
41|     }
42|
43|     while(1) {
44|         sin_size = sizeof(struct sockaddr);
45|         if ((recv_size = recvfrom(sockfd, buffer, BUF_SIZE, 0, (struct sockaddr *)&
46|             client_addr, &sin_size)) == 0) {
47|             close(sockfd);
48|             if (errno != 0) {
49|                 perror("recvfrom");
50|             }
51|             break;
52|         }
53|         buffer[recv_size] = '\0';
54|         printf("Received message. Size: %d, message: %s", recv_size, buffer);
55|         if (sendto(sockfd, buffer, recv_size, 0, (struct sockaddr *)&client_addr,
56|             sizeof(struct sockaddr)) == -1) {
57|             perror("sendto");
58|             close(sockfd);
59|             break;
}
}

```

Listing 4.34: Serwer ECHO w C

Pierwsza zmiana to zmiana napisu w linii 20. Teraz nasz program to `udp_server`, a nie `tcp_server`. Właściwie, to moglibyśmy skorzystać z wartości `argv[0]`, która przechowuje faktyczną nazwę naszego skryptu, bo przecież nie możemy zbroić nikomu zmiany nazwy pliku. Dalsze zmiany mają już wymiar techniczny.

W linii 28 tworzymy gniazdo datagramowe, a nie strumieniowe, czyli jako drugi argument podajemy `SOCK_DGRAM`.

Dalej nie używamy funkcji `listen` i `accept`, ponieważ nie będziemy oczekiwali na połączenia, a na same pakiety. Stąd, nie mamy dwóch pętli nieskończonych (jedna na połączenia, a druga na pakiety z danego połączenia), tylko jedną pętlę nieskończoną w linii 42 na wszystkie pakiety.

W gniazdach strumieniowych funkcja `accept` przymywała połączenie, a mówiąc bardziej technicznie, przeprowadzała z klientem procedurę *3-Way Handshake*. W gniazdach datagramowych tego nie ma.

Oczywiście musimy wiedzieć skąd pakiety przyszły, a skoro nie ma `accept`, to inna funkcja musi nam o tym mówić. W linii 44 używamy funkcji `recvfrom` (zamiast `recv`), która oprócz deskryptora gniazda, danych `buffer`, maksymalnej długości pobranych danych `BUF_SIZE` oraz flag (w tym przypadku żadnej

z nich) przyjmuje jeszcze dwa parametry wyjściowe, czyli takie które uzupełni. Pierwszy z nich to wskaźnik na wcześniej stworzoną strukturę `client_addr`, zaś drugi to parametr wejściowo-wyściowy, który jest wskaźnikiem na liczbę całkowitą. Przed wywołaniem funkcji należy wartość ostatniego argumentu `sin_size` ustalić na rozmiar struktury z poprzedniego argumentu (co czynimy w linii 43), zaś po wywołaniu funkcji pod tym adresem będzie rozmiar faktycznie uzupełnionej struktury (zwykle wartość tego argumentu się nie zmienia). A zatem, po wywołaniu funkcji `recvfrom`, w zmiennej `client_addr` będziemy mieli adres, z którego nadszedł pakiet.

Nie tworzymy nowego połączenia, więc nie jest tworzone żadne nowe gniazdo, a odpowiedź wysyłamy za pomocą tego samego gniazda. W linii 53 korzystamy z funkcji `sendto`, która oprócz deskryptora gniazda, danych oraz ich rozmiaru (parametry funkcji `send`) przyjmuje jeszcze adres, pod który wysłać pakiet. Adres ten definiowany jest przed 2 dodatkowe parametry. Pierwszy z nich to wskaźnik na strukturę z określonym adresem, zaś drugi to rozmiar tej struktury. W naszym przykładzie wykorzystujemy adres, z którego przyszedł poprzedni pakiet (linia 44), stąd możemy nowy pakiet nazwać odpowiedzią.

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4 #include <errno.h>
5 #include <sys/types.h>
6 #include <sys/socket.h>
7 #include <netinet/in.h>
8 #include <arpa/inet.h>
9 #include <unistd.h>
10
11 int main(int argc, char * argv[]) {
12     int BUF_SIZE = 1024;
13     char * ip_addr;
14     int port;
15     int sockfd;
16     struct sockaddr_in server_addr;
17     int recv_size, read_size, sent_size;
18     char buffer[BUF_SIZE+1];
19
20     if (argc != 3) {
21         fprintf(stderr, "usage: udp_client ip port\n");
22         exit(1);
23     }
24     ip_addr = argv[1];
25     port = atoi(argv[2]);
26     if (port <= 0) {
27         fprintf(stderr, "error: invalid port\n");
28         exit(1);
29     }
30     if ((sockfd = socket(AF_INET, SOCK_DGRAM, 0)) == -1) {
31         perror("socket");
32         exit(1);
33     }
34
35     server_addr.sin_family = AF_INET;
36     server_addr.sin_port = htons(port);
37     inet_aton(ip_addr, &server_addr.sin_addr);
38
39     while(1) {
40         printf("Data to send (Ctrl+D quits): ");
41         if (fgets(buffer, BUF_SIZE, stdin) == NULL) {
42             close(sockfd);

```

```

43     perror("fgets");
44     exit(1);
45 }
46 read_size = strlen(buffer);
47
48 if ((sent_size = sendto(sockfd, buffer, read_size, 0, (struct sockaddr *)&
49     server_addr, sizeof(struct sockaddr))) == -1) {
50     close(sockfd);
51     perror("sendto");
52     exit(1);
53 }
54 if ((recv_size = recvfrom(sockfd, buffer, BUF_SIZE, 0, NULL, NULL)) == 0) {
55     close(sockfd);
56     if (errno != 0) {
57         perror("recv");
58         exit(1);
59     }
60     break;
61 }
62 buffer[recv_size] = '\0';
63 printf("Answer: %s", buffer);
64
65 }
66 }
```

Listing 4.35: Klient ECHO w C

W kliencie również zmieniamy nazwę programu na `udp_client`.

Podobnie, jak w przypadku serwera tworzymy gniazdo datagramowe, a nie strumieniowe, czyli w linii 30 drugi argument zmieniamy na `SOCK_DGRAM`.

Jak wspomniałem w przypadku serwera, gniazdo datagramowe nie nawiązuje połączenia, dlatego nie używamy funkcji `connect`. W zamian, po wczytaniu danych od użytkownika, przekazujemy do metody `sendto` (linia 48) zarówno deskryptor gniazda, dane do wysłania, ich rozmiar oraz flagi, jak również wskaźnik na strukturę zawierającą adres docelowy oraz rozmiar tej struktury (wcześniej były one wykorzystywane w funkcji `connect`). Oczywiście adres ten może się zmieniać, podczas gdy w przypadku gniazd strumieniowych wszystkie dane wysyłane za pomocą jednego strumienia miały tego samego adresata.

Następnie w linii 54 pobieramy dane z *jakiegoś* adresu i w linii 64 wyświetlamy je na ekranie. Celowo dodałem słowo *jakiegoś*, ponieważ nie mamy pewności, że pobrane dane pochodzą z tego samego adresu, na który wcześniej wysłaliśmy dane (w gniazdach strumieniowych mamy taką pewność). Stąd, jeśli chciałbym mieć pewność, że to pakiet z odpowiedzią to powinieneś porównać adres docelowy z linii 48 z adresem nadawcy z linii 54. Jednakże zakładam, że mój klient nie będzie otrzymywał pakietów z innych miejsc, niż serwer, na który wysłał wiadomość, dlatego jako argument przekazuje wartości `NULL`, co znaczy, że nie interesuje mnie, skąd pakiet przyszedł.

W takim razie kiedy używać gniazd połączeniowych (TCP), a kiedy bezpołączeniowych (UDP)?

To pytanie można uprościć do pytania: Kiedy używać TCP, a kiedy UDP. Ciekawa analiza znajduje się pod adresem: http://www.differencetech.com/difference/TCP_vs_UDP.

Krótko mówiąc, gdy zależy nam na niezawodności, czyli na wykrywaniu wszelkich błędów transmisji, zagwarantowaniu dotarcia oraz porządku przyjmowania pakietów, a mniej istotna jest dla nas wydajność, to powinniśmy użyć TCP.

Natomiast jeżeli zależy nam na bardzo dużej wydajności lub protokół polega na odpowiadaniu bardzo dużej liczbie klientów niedużej wielkości pakietami, to powinniśmy użyć UDP. Co więcej, UDP może być wykorzystany do komunikacji typu broadcast lub multicast, podczas gdy TCP, ze względu na utrzymywane połączenie między 2 stronami, nie umożliwia takiej komunikacji.

Jednym z przykładów, w których najtrudniej się zdecydować na protokół są gry MMO.

Przykładami protokołów wykorzystujących TCP są HTTP, HTTPS, FTP, Telnet, zaś UDP to DNS, DHCP, TFTP, VoIP.

4.1.6 Protokół IP w wersji 6

TBC

4.2 Protokół poczty

Definicji poczty e-mail przytaczać nie trzeba, jednakże warto przypomnieć sobie jej podstawowe techniczne elementy. Do obsługi poczty wykorzystywane są protokoły odpowiedzialne za wysyłanie poczty (SMTP) oraz za jej odbieranie (POP3, IMAP). W tym rozdziale przyjrzymy się dokładniej tym protokołom oraz zbudujemy aplikacje, które będą komunikowały się z serwerem poczty.

4.2.1 Wysyłanie wiadomości e-mail

Kiedy wysyłamy wiadomość e-mail z serwera, na którym jest nasza skrzynka, nie jest ona przesyłana bezpośrednio na serwer, na którym jest skrzynka odbiorcy. Wiadomość wędruje przez wiele serwerów pocztowych, na których uruchomiony jest program MTA (ang. mail Transfer Agent), którego zadaniem jest przekazanie wiadomości do następnego serwera na podstawie nagłówków.

Programy MTA wykorzystują protokół SMTP (ang. Simple Mail Transfer Protocol), który służy tylko do wysyłania wiadomości, czyli de facto przekazania jej z serwera nadawcy do serwera odbiorcy.

Naszym zadaniem będzie napisanie programu pocztowego, który połączy się z serwerem za pomocą protokołu SMTP i wyśle wiadomość. Oczywiście istnieją biblioteki, które większość obsługi protokołu realizują za nas, jednakże my będziemy chcieli zrealizować ten sam cel bez bibliotek.

Od czego zaczniemy? Zaczniemy od końca, czyli od przykładu prostego programu w Pythonie do wysyłania wiadomości, wykorzystującego biblioteki

smtplib oraz email. I to od razu wyślemy wiadomość z załącznikiem.

```

1  from os.path import basename
2  import smtplib
3  from email.mime.image import MIMEImage
4  from email.mime.multipart import MIMEMultipart
5  from email.mime.text import MIMEText
6  from email.mime.application import MIMEApplication
7
8
9
10 def send_email(sender, recipient, subject, message, attachments):
11     msg = MIMEMultipart()
12     msg['To'] = recipient
13     msg['From'] = sender
14     msg['Subject'] = subject
15
16     part = MIMEText('text', "plain")
17     part.set_payload(message)
18     msg.attach(part)
19
20     for f in attachments:
21         with open(f, "rb") as fd:
22             part = MIMEApplication(
23                 fd.read(),
24                 Name=basename(f)
25             )
26             part['Content-Disposition'] = 'attachment; filename="%s"' % basename(f)
27             msg.attach(part)
28
29     session = smtplib.SMTP('127.0.0.1', 25)
30     # session.set_debuglevel(1)
31     session.ehlo()
32     session.sendmail(sender, recipient, msg.as_string())
33     print("Your email is sent to {0}.".format(recipient))
34     session.quit()
35
36 if __name__ == '__main__':
37     sender = raw_input("What is the sender's email address? ")
38     recipient = raw_input("What is the recipient's email address? ")
39     subject = raw_input("What is the subject? ")
40
41     message = ''
42     msgline = raw_input('Enter the message. Empty line ends the message.\n')
43     while msgline != '':
44         message += msgline + '\n'
45         msgline = raw_input('')
46
47     attachments = []
48     att = raw_input('Enter the file names of attachments. Empty line ends the list.\n')
49     while att != '':
50         attachments.append(att)
51         att = raw_input('')
52
53     send_email(sender, recipient, subject, message, attachments)

```

Listing 4.36: Wysyłanie maila przez serwer lokalny

Skrypt zaczyna się od funkcji `send_email`, do której wróćmy na chwilę, bo pierwsze instrukcje, które zostaną uruchomione to te, które zaczynają się od 37 linii. Tam prosimy o adres e-mail nadawcy (tak, możemy go ustalać!), adres odbiorcy, temat, treść wiadomości oraz listę załączników. Wszystkie te zmienne przekazywane są do funkcji w linii 10.

Funkcja na początku tworzy wiadomość `Multipart`, co oznacza, że będzie

ona składała się z kilku części. Można również wysłać sam tekst bez formatu MIME, jednakże takie wiadomości nie mogą zawierać załączników, ani wersji HTML, która uwzględnia formatowanie podczas wyświetlania wiadomości w przeglądarce.

W liniach 12 - 14 ustawiane są nagłówki To, From oraz Subject. Następnie, w linii 16, tworzona jest pierwsza część wiadomości o typie text/plain. Jej dane (ang. payload) to treść wiadomości, ustawiana w linii 17, zaś w linii 18 część ta jest dołączana do wiadomości.

Następnie w pętli for, dla każdego pliku z listy załączników, tworzona jest część typu application, której wartością jest treść pliku, a atrybut Name jest ustawiany na nazwę pliku. Dodatkowo, w części ustalany jest nagłówek Content-Disposition, w którym zaznaczono, że jest to załącznik do wiadomości, a jego nazwa jest taka sama, jak nazwa wczytanego pliku. Na końcu, w linii 27, część application dołączana jest do wiadomości.

Dalej w linii 29 skrypt łączy się z lokalnym serwerem poczty na porcie 25 (zakładam, że korzystasz z kontenera Dockera), przesyła wiadomość HELO (o niej trochę więcej powiemy później) i korzystając z metody sendmail wysyła przygotowaną wiadomość przekonwertowaną do napisu. To wszystko.

Na Rysunku 4.3 znajduje się zrzut ekranu z uruchomienia skryptu. Jednakże to co nas bardziej interesuje to dane, którymi za pomocą gniazd wymieniły się serwer i klient.

```
root@a37289c3b545:/opt/pas/email# python send_email.py
What is the sender's email address? [REDACTED]@gmail.com
What is the recipient's email address? [REDACTED]@gmail.com
What is the subject? Test
Enter the message. Empty line ends the message.
TEST1
TEST2

Enter the file names of attachments. Empty line ends the list.
send_email.py

You email is sent to [REDACTED]@gmail.com.
root@a37289c3b545:/opt/pas/email#
```

Rysunek 4.3: Wysyłanie maila - uruchomienie skryptu.

W skrypcie linia 30 jest zakomentowana. Zawiera ona wywołanie metody set_debuglevel z wartością 1. Oznacza ona, że na ekranie będą wyświetlane wszystkie komunikaty, w tym wiadomości wymieniane między klientem (skrypte) i serwerem. Odkomentuj ją i uruchom skrypt ponownie. Na ekranie pojawi się wiadomość podobna do poniższej.

```
1 root@a37289c3b545:/opt/pas/email# python send_email.py
2 What is the sender's email address? <sender>@gmail.com
3 What is the recipient's email address? <receiver>@gmail.com
4 What is the subject? Wiadomosc testowa
5 Enter the message. Empty line ends the message.
6 Wiadomosc jawnia.
7
8 Enter the file names of attachments. Empty line ends the list.
9 wiadomosc
```

```

10| send: 'ehlo [172.17.0.2]\r\n'
11| reply: '250-a37289c3b545 Hello localhost [127.0.0.1], pleased to meet you\r\n'
12| reply: '250-ENHANCEDSTATUSCODES\r\n'
13| reply: '250-PIPELINING\r\n'
14| reply: '250-EXPN\r\n'
15| reply: '250-VERB\r\n'
16| reply: '250-8BITMIME\r\n'
17| reply: '250-SIZE\r\n'
18| reply: '250-DSN\r\n'
19| reply: '250-ETRN\r\n'
20| reply: '250-AUTH DIGEST-MD5 CRAM-MD5\r\n'
21| reply: '250-DELIVERBY\r\n'
22| reply: '250-HELP\r\n'
23| reply: retcode (250); Msg: a37289c3b545 Hello localhost [127.0.0.1], pleased to
   meet you
24| ENHANCEDSTATUSCODES
25| PIPELINING
26| EXPN
27| VERB
28| 8BITMIME
29| SIZE
30| DSN
31| ETRN
32| AUTH DIGEST-MD5 CRAM-MD5
33| DELIVERBY
34| HELP
35| send: 'mail FROM:<sender@gmail.com> size=2671\r\n'
36| reply: '250 2.1.0 <sender@gmail.com>... Sender ok\r\n'
37| reply: retcode (250); Msg: 2.1.0 <sender@gmail.com>... Sender ok
38| send: 'rcpt TO:<receiver@gmail.com>\r\n'
39| reply: '250 2.1.5 <receiver@gmail.com>... Recipient ok\r\n'
40| reply: retcode (250); Msg: 2.1.5 <receiver@gmail.com>... Recipient ok
41| send: 'data\r\n'
42| send: 'data\r\n'
43| reply: '354 Enter mail, end with "." on a line by itself\r\n'
44| reply: retcode (354); Msg: Enter mail, end with "." on a line by itself
45| data: (354, 'Enter mail, end with "." on a line by itself')
46| send: 'Content-Type: multipart/mixed; boundary
      =====8001626673766582685==\r\nMIME-Version: 1.0\r\nTo: <receiver
      >@gmail.com\r\nFrom: <sender>@gmail.com\r\nSubject: Wiadomosc testowa\r\n\r\n
      =====8001626673766582685==\r\nContent-Type: text/plain; charset="us-ascii"\r\nMIME-Version: 1.0\r\nContent-Transfer-Encoding: 7bit\r\n\r\n
      nWiadomosc jawnna.\r\n\r\n=====8001626673766582685==\r\nContent-
      Type: application/octet-stream; Name="wiadomosc"\r\nMIME-Version: 1.0\r\n
      Content-Transfer-Encoding: base64\r\nContent-Disposition: attachment;
      filename="wiadomosc"\r\n\r\nU1VQRVIGVEFKTkEgV01BRE9NT1NDISEh\r\n
      =====8001626673766582685==\r\n\r\n'
47| reply: '250 2.0.0 u8RIF2Ba000154 Message accepted for delivery\r\n'
48| reply: retcode (250); Msg: 2.0.0 u8RIF2Ba000154 Message accepted for delivery
49| data: (250, '2.0.0 u8RIF2Ba000154 Message accepted for delivery')
50| You email is sent to <receiver>@gmail.com.
51| send: 'quit\r\n'
52| send: '221 2.0.0 a37289c3b545 closing connection\r\n'
53| reply: retcode (221); Msg: 2.0.0 a37289c3b545 closing connection

```

Listing 4.37: Logi z wysyłania wiadomości

Protokół SMTP jest protokołem tekstowym, to znaczy że każda komenda jest przesyłana w formie tekstu i jest zrozumiała dla człowieka. Często, o czym przekonamy się później, protokoły są nieczytelne, ponieważ komendy są wartościami liczbowymi (np. pierwszy bajt wiadomości to typ komendy) i wtedy trzeba pisać dekodery, które zamieniają nieczytelne bajty na czytelne komuni-

katy.

Na tym listingu znajduje się to co na Rysunku 4.3, jak również wyświetlona jest pełna komunikacja z serwerem. Widać na przykład, że pierwsza wiadomość wysyłana przez klienta to `ehlo [172.17.0.2]\r\n` (znaki `\r` i `\n` są istotne), po czym serwer odpowiada kodem 250.

Ponadto, w linii 46, widać jak wygląda wiadomość e-mail, składająca się z wielu części, po skonwertowaniu na napis. **Czy jesteś w stanie powiedzieć co jest przesyłane w załączniku?**

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	172.17.0.2	87.246.207.237	SMTP	112	C: rcpt to: damian.rusinek@poczta.umcs.lublin.pl
2	0.057133	87.246.207.237	172.17.0.2	TCP	66	25-36742 [ACK] Seq=1 Ack=47 Win=114 Len=0 TStamp=2
3	0.063729	87.246.207.237	172.17.0.2	SMTP	80	S: 250 Accepted
4	0.063750	172.17.0.2	87.246.207.237	TCP	66	36742-25 [ACK] Seq=47 Ack=15 Win=237 Len=0 TStamp=2
						Simple Mail Transfer Protocol
						Frame 1: 112 bytes on wire (896 bits), 112 bytes captured (896 bits)
						► Ethernet II, Src: 02:42:ac:11:00:02 (02:42:ac:11:00:02), Dst: 02:42:25:90:a2:1f (02:42:25:90:a2:1f)
						► Internet Protocol Version 4, Src: 172.17.0.2 (172.17.0.2), Dst: 87.246.207.237 (87.246.207.237)
						► Transmission Control Protocol, Src Port: 36742 (36742), Dst Port: 25 (25), Seq: 1, Ack: 1, Len: 46
						Simple Mail Transfer Protocol
						Command Line: rcpt to: damian.rusinek@poczta.umcs.lublin.pl\r\n
						Command: rcpt
						Request parameter: to: damian.rusinek@poczta.umcs.lublin.pl
						0000 02 42 25 90 a2 1f 02 42 ac 11 80 02 88 88 45 88 .B8...BE.
						0010 00 62 14 50 40 00 40 06 52 4f ac 11 00 02 57 f6 .b.P@.R0...W.
						0020 cf ed 8f 86 00 19 d1 1a 11 38 8c 72 f1 13 88 18 ..K.....8.F..
						0030 00 ed d4 4b 00 00 01 01 08 0a 00 96 22 c5 0d 84 ..K.....8.F..
						0040 89 24 72 63 70 74 20 74 6f 3a 20 64 61 6d 69 61 .rcpt t o: damia
						0050 6e 2e 72 75 73 69 6e 65 6b 40 70 6f 63 7a 74 61 n.rusine @poczta
						0060 2e 75 6d 63 73 2e 6c 75 62 66 69 6e 2e 70 6c 6a .umcs.lu blin.pl.

Rysunek 4.4: Wysyłanie maila - podsłuchana komunikacja.

Poznaliśmy zatem pierwsze miejsce, z którego możemy czerpać wiedzę o formacie wymienianych komunikatów. Drugie miejsce to skorzystanie z dobrze już nam znanego Wiresharka. Wystarczy, że w kontenerze uruchomisz `tcpdump -i eth0 -w out.pcap` (interfejs może się różnić), po czym w drugim terminalu uruchomisz nasz skrypt. Po zakończeniu wysyłania wiadomości wróć do pierwszego terminala i zakończ działanie `tcpdump` (`Ctrl+C`). Pojawi się plik `out.pcap`, który możesz otworzyć w Wiresharku i również przeglądać przesyłane pakiety. Jeden z nich przedstawiam na Rysunku 4.4. Ustala on adres odbiorcy.

Ostatnim i zarazem najbardziej pewnym i obszernym źródłem informacji o protokole SMTP jest dokument <https://www.ietf.org/rfc/rfc2821.txt>. Na jego przykładzie omówimy sobie najprostszy scenariusz wysyłania maila. W rozdziale 3 pt. "The SMTP Procedures: An Overview" mamy opis kolejnych kroków wykonywanych podczas komunikacji klienta z serwerem SMTP.

Pierwszym krokiem *Session Initiation* jest połączenie się z serwerem poczty, który odpowiada wiadomością powitalną. Każda wiadomość z serwera zawiera na początku kod odpowiedzi, który jest liczbą naturalną. Ponadto serwer może wysłać wiadomość wielowierszową, lecz w każdej, oprócz ostatniej, linii po kodzie powinien być znak „-”. W ostatniej linii znakiem tym powinna być spacja.

Wiadomość powitalna z serwera zawiera kod 220, za którym może być wiadomość opisująca serwer (np. domena serwera, wersja oprogramowania).

Na każdą wiadomość klienta, w momencie wystąpienia błędu, serwer odpowiada wiadomością z kodem błędu. Na przykład zamiast wiadomości 220, serwer może odpowiedzieć kodem 554, gdy nie zezwala na wysłanie wiadomości.

Kolejny krok *Client Initiation* to wysłanie przez klienta wiadomości EHLO (lub HELO, gdy klient lub serwer nie obsługuje rozszerzeń) po której występuje identyfikator klienta (np. hostname). W odpowiedzi serwer odsyła wiadomość z kodem 250, własną domeną oraz informacjami o kliencie.

Od tej chwili klient definiuje wiadomość e-mail do wysłania, czyli rozpoznaje krok *Mail Transactions*. Pierwsza komenda to MAIL FROM:<source><CRLF>. Wartość <source> to skrzynka źródłowa, definiująca nadawcę wiadomości. Wartość <CRLF> to znaki \r\n, czyli Windowsowe znaki nowej linii (na Linuxie jest tylko \n). W odpowiedzi serwer przesyła wiadomość 250 OK.

Druga komenda to RCPT TO:<forward-path> <CRLF>, która ustala adres odbiorcy w formie nazwy skrzynki lub pełnego adresu e-mail. W sytuacji, gdy serwer uzna adres za nieosiagalny, zwróci wiadomość z kodem 554. Jeśli serwer akceptuje adres e-mail odbiorcy, zwraca wiadomość 250 OK. Komenda może być powtórzona wielokrotnie, dzięki czemu wiadomość może być wysłana do wielu odbiorców.

Ostatnią komendą definiującą wiadomość jest DATA <CRLF>, na którą serwer odpowiada kodem 354 i oczekuje na nagłówki i treść wiadomości e-mail. Każda kolejna przesłana linia to część nagłówka lub treści wiadomości. Dopiero linia „..<CRLF>” kończy treść wiadomości i po jej przesłaniu serwer odpowiada 250 OK, co skutkuje wysłaniem wiadomości e-mail.

Przykład treści wiadomości znajduje się na Listingu 4.37 w linii 46. Jest to przykład wiadomości składającej się z wielu części, w których można umieścić np. załączniki. Więcej informacji o rozszerzeniu MIME możesz znaleźć pod tym adresem <https://www.ietf.org/rfc/rfc1341.txt>. Jeśli chcesz wysłać prostą wiadomość bez załączników, to możesz nie korzystać z MIME i wtedy Twoja wiadomość będzie wyglądać jak pierwsza część z Listingu 4.37. Pierwsze wiersze to nagłówki, takie jak From: <adres-nadawcy>, To: <adres-nadawcy>, Cc: <adres-nadawcy>, itp. zakończone <CRLF>. Po nich jest pusta linia <CRLF>, a po niej kolejne linie aż do znaku końca wiadomości (kropka) to treść wiadomości e-mail, która zostanie wyświetlona odbiorcy. Na poniższym listingu przedstawiono nagłówki oraz treść wiadomości wysłanej w ramach połączenia z Listingiem 4.37, lecz bez wykorzystania MIME.

```

1 To: <receiver>@gmail.com\r\n
2 From: <sender>@gmail.com\r\n
3 Subject: Wiadomosc testowa\r\n
4 \r\n
5 Wiadomosc jawnego.\r\n
6 .\r\n

```

Listing 4.38: Nagłówki i treść wiadomości

Każdy nagłówek oraz wiersz treści zostało umieszczone w odrębnej linii dla czytelności, jednakże faktycznie byłaby to jedna linia, tak jak na Listingu 4.37 w linii 46, ponieważ przejście do nowej linii zastąpiono znakami \r\n.

Po potwierdzeniu wysłania wiadomości e-mail klient powinien wysłać komendę zakończenia połączenia QUIT<CRLF>, na którą serwer odpowiada kodem 221.

Po tym krótkim opisie SMTP możemy już napisać klienta SMTP, który wyśle prostą wiadomość bez załączników, którego kod umieściłem na Listingu 4.39.

```

1 import sys
2 import socket
3
4 BUF_SIZE = 1024
5
6 def get_resp_code(line):
7     return int(line[:3])
8
9 if __name__ == "__main__":
10
11     if len(sys.argv) != 3:
12         sys.stderr.write("usage: %s ip port\n", sys.argv[0])
13         exit(1)
14
15     try:
16         addr = sys.argv[1]
17         port = int(sys.argv[2])
18         assert port > 0
19     except:
20         sys.stderr.write("error: invalid port\n")
21         exit(1)
22
23     sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
24     try:
25         sender_email = raw_input('Sender e-mail:')
26
27         print 'Please provide receivers\' e-mails in separate lines (empty line ends
28             list):'
29         receiver_emails = []
30         receiver_email = raw_input('')
31         while receiver_email != '':
32             receiver_emails.append(receiver_email)
33             receiver_email = raw_input('')
34
35         subject = raw_input('Subject:')
36
37         msg = ''
38         msg += 'From: %s\r\n' % (sender_email,)
39         for receiver_email in receiver_emails:
40             msg += 'To: %s\r\n' % (receiver_email,)
41         msg += 'Subject: %s\r\n' % (subject,)
42         msg += '\r\n'
43
44         print 'Please provide message (empty line ends message):'
45         msg_line = raw_input('')
46         while msg_line != '':
47             msg += msg_line + '\r\n'
48             msg_line = raw_input('')
49             msg += '.\r\n'
50
51         sock.connect((addr, port))
52         data = sock.recv(BUF_SIZE)
53         code = get_resp_code(data)
54         if code != 220:
55             print 'Cannot connect to server.'
56             exit(1)
57
58         sock.send('HELO example.com\r\n')
59         data = sock.recv(BUF_SIZE)
60         code = get_resp_code(data)

```

```

60|     if code != 250:
61|         print 'Error on HELO command:', data[4:]
62|         exit(1)
63|
64|     sock.send('MAIL FROM: %s\r\n' % (sender_email,))
65|     data = sock.recv(BUF_SIZE)
66|     code = get_resp_code(data)
67|     if code != 250:
68|         print 'Error on MAIL FROM command:', data[4:]
69|         exit(1)
70|
71|     for receiver_email in receiver_emails:
72|         sock.send('RCPT TO: %s\n' % (receiver_email,))
73|         data = sock.recv(BUF_SIZE)
74|         code = get_resp_code(data)
75|         if code != 250:
76|             print 'Error on RCPT TO command:', data[4:]
77|             exit(1)
78|
79|     sock.send('DATA\r\n')
80|     data = sock.recv(BUF_SIZE)
81|     code = get_resp_code(data)
82|     if code != 354:
83|         print 'Error on DATA command:', data[4:]
84|         exit(1)
85|
86|     sock.send(msg)
87|     data = sock.recv(BUF_SIZE)
88|     code = get_resp_code(data)
89|     if code != 250:
90|         print 'Error on message:', data[4:]
91|         exit(1)
92|
93|     sock.send('QUIT\r\n')
94|     data = sock.recv(BUF_SIZE)
95|     code = get_resp_code(data)
96|     if code != 221:
97|         print 'Error on QUIT command:', data[4:]
98|         exit(1)
99|
100|    print 'E-mail sent.'
101|    sock.close()
102|
103|except socket.error, e:
104|    print 'Error:', e

```

Listing 4.39: Klient SMTP

W linii 6 i 7 znajduje się funkcja `get_resp_code`, która zamienia 2 pierwsze znaki na wartość liczbową, czyli odczytuje kod odpowiedzi.

Dalej do linii 23 sprawdzamy poprawność argumentów programu i tworzymy gniazdo strumieniowe. W liniach 25-43 pobieramy od użytkownika adres nadawcy, adresy odbiorców oraz temat wiadomości. W liniach 36-41 tworzymy nagłówek wiadomości e-mail zgodnie z tym co opisałem wyżej, czyli dodajemy nagłówki `From`, `To` (tych jest kilka, bo adresatów może być kilku) oraz `Subject`. Wszystkie linie zakończone są znakami `<CRLF>`, a na końcu dodana jest jeszcze jedna pusta linijka z tymi znakami.

Dalej pytamy użytkownika o treść wiadomości. Dopóki wczytana linia nie jest pusta, czyli dopóki użytkownik coś wpisuje, dodajemy tę linię do wiadomości, a jak poda linię pustą, to dodajemy na koniec znaki `.<CRLF>`, kończące treść wiadomości.

W tej chwili mamy gotowe dane do wysłania do serwera, więc nawiązujemy z nim kontakt w linii 50 i odbieramy wiadomość powitalną. Po odebraniu każdej z wiadomości sprawdzamy jej kod i jeśli jest to kod błędu, wyświetlamy informację i kończymy działanie programu (właściwie to powinniśmy jeszcze wysłać komendę QUIT i zamknąć gniazdo, ale dla czytelności pominąłem ten punkt).

Kolejny krok to komenda HELO z dowolną domeną (np. `example.com`), komenda MAIL FROM z adresem nadawcy oraz komendy RCPT TO dla adresu każdego z odbiorców. Po każdej z tych oraz kolejnych komend wysyłamy `<CRLF>`. Teraz możemy przesłać treść wiadomości, więc zaczynamy w linii 79 od komendy DATA, po czym wysyłamy wcześniej przygotowaną wiadomość, zapisaną w zmiennej msg (z kropką w ostatniej linii). Jeśli wszystko poszło poprawnie, to możemy zakończyć proces wysyłając komendę QUIT i zamknąć gniazdo.

Dalsze kroki...

- Istnieje rozszerzenie protokołu SMTP wykorzystujące szyfrowanie TLS i jest ono opisane w dokumencie <https://www.ietf.org/rfc/rfc3207.txt>. Zastanów się, w jaki sposób dodać do naszego klienta obsługę TLS.
- Spróbuj napisać serwer poczty, obsługujący protokół SMTP. Nie realizuj faktycznego wysyłania e-maila, tylko zasymuluj jego działanie tak, żeby nasz prosty klient SMTP myślał, że wiadomość została wysłana.

4.2.2 Pobieranie wiadomości e-mail

Do odczytywania wiadomości ze swojej skrzynki pocztowej można skorzystać z jednego z protokołów POP3 i IMAP. W przypadku protokołu POP3 wszystkie wiadomości ze skrzynki pocztowej pobierane są w całości do skrzynki lokalnej na komputerze oraz pozostawione lub usunięte z serwera. Dzięki takiemu rozwiązaniu możliwa jest późniejsza praca z wiadomościami e-mail bez połączenia z Internetem. Analogicznie możliwe jest tworzenie nowych wiadomości do wysłania, które zostaną wysłane podczas następnego połączenia z serwerem poczty. Zatem plusem protokołu jest jednorazowe pobranie zawartości skrzynki i możliwa praca w trybie off-line, natomiast minusem jest przynajmniej moment pobrania wiadomości do skrzynki lokalnej, co może trochę potrwać w przypadku wiadomości z dużymi załącznikami. Ponadto, jeśli po pobraniu wiadomości będą one usuwane z serwera, a mamy skonfigurowane skrzynki na kilku komputerach, to na każdym z nich będzie część wiadomości.

Protokół IMAP charakteryzuje się zgoła innym podejściem do zarządzania skrzynką pocztową. Na skrzynkę lokalną na komputerze pobierane są jedynie nagłówki wiadomości, dzięki czemu transfer jest znacznie mniejszy. Jeśli interesuje nas jakaś wiadomość, to po jej wybraniu pobierana jest w całości. Ponadto, protokół IMAP pozwala na grupowanie wiadomości w różnych folderach, gdzie domyślnym folderem jest `Inbox`. Zatem wykorzystanie protokołu

IMAP pozwala zmniejszyć transfer danych, jednakże wymaga ciągłego dostępu do Internetu.

W tej części skryptu skorzystamy z protokołu IMAP i, podobnie jak w części poprzedniej, najpierw napiszemy program wykorzystując bibliotekę obsługującą protokół IMAP, po czym napiszemy prostego klienta wykorzystując gniazda.

```

1 import imaplib
2 import getpass
3
4 IMAP_SERVER_ADDR = '127.0.0.1'
5 IMAP_SERVER_PORT = 143
6
7 def show_emails(server, port, username, password):
8     mailbox = imaplib.IMAP4(server, port)
9     resp, msg = mailbox.login(username, password)
10    if resp != 'OK':
11        print 'Auth error: %s' % msg[0]
12        exit(1)
13    resp, msg = mailbox.select('Inbox')
14    if resp != 'OK':
15        print 'Select mailbox error: %s' % msg[0]
16        exit(1)
17
18    resp, data = mailbox.search(None, 'ALL')
19
20    for num in data[0].split():
21        resp, msg = mailbox.fetch(num, '(RFC822)')
22        resp, text = mailbox.fetch(num, '(BODY[TEXT])')
23        print 'Message no. %s\n' % (num,)
24        print 'Message body:'
25        print text[0][1]
26        print 'Whole message:'
27        print msg[0][1]
28        print '-----'
29
30    mailbox.close()
31    mailbox.logout()
32
33 if __name__ == '__main__':
34     username = raw_input("Login: ")
35     password = getpass.getpass(prompt="Password:")
36
37 show_emails(IMAP_SERVER_ADDR, IMAP_SERVER_PORT, username, password)

```

Listing 4.40: Pobieranie mailu ze skrzynki

Po wcześniejszym połączeniu się z lokalnym serwerem poczty i wysłaniu wiadomości e-mail z adresu `test@localhost` na ten sam adres i uruchomieniu skryptu w konsoli pojawił się wynik przedstawiony na Rysunku 4.5. Wszystkie te operacje wykonywane były na kontenerze Dockera, odpowiednio już skonfigurowanym. Możesz je powtórzyć (na kontenerze jest już użytkownik `test` z hasłem `test`).

Skrypt rozpoczyna swoje działanie w linii 34, w której pyta użytkownika o jego login (tutaj użyłem `test`). W linii następnej korzystam z biblioteki `getpass`, która umożliwia pobranie danych od użytkownika bez ich wyświetlenia (nie chcemy widzieć naszego hasła na ekranie).

Skrypt kończy się wywołaniem funkcji `show_emails`, do której przekazywane są stałe z adresem serwera (`localhost`), portem 143 oraz nazwą i hasłem pobranymi od użytkownika. Protokół IMAP jest rozdzielony, podob-

```
root@0f364387f404:/opt/pas/email# python get_emails.py
Login: test
Password:
Message no. 1

Message body:
aaa.

Whole message:
Return-Path: <test@localhost>
X-Original-To: test@localhost
Delivered-To: test@localhost
Received: from a (localhost [127.0.0.1])
    by c944455e5c4d (Postfix) with SMTP id 0EABF90B
    for <test@localhost>; Wed,  5 Oct 2016 14:33:11 +0000 (UTC)
Subject: aaa
Message-Id: <20161005143320.0EABF90B@c944455e5c4d>
Date: Wed,  5 Oct 2016 14:33:11 +0000 (UTC)
From: test@localhost

aaa.

-----

```

Rysunek 4.5: Pobranie maili (IMAP) - uruchomienie skryptu.

nie jak SMTP, na wersję szyfrowaną i nie szyfrowaną. Pod portem 143 kryje się wersja nieszyfrowana i z niej korzystamy, ponieważ nie poznaliśmy jeszcze metody na ustanowienie szyfrowanego połączenia. Wersja szyfrowana IMAPa zwykle wystawiana jest na porcie 993.

W funkcji `show_emails` tworzony jest klient IMAP, któremu w konstruktorze przekazywany jest adres serwera oraz port usługi. Następnie wywoływana jest metoda `login` z danymi przekazanymi przez użytkownika. W linii 10 weryfikowana jest odpowiedź serwera, która po pomyślnym wykonaniu komendy zwraca odpowiedź `OK`. W przeciwnym razie wyświetlany jest komunikat błędu, zwracany przez serwer razem z odpowiedzią.

Protokół IMAP wspiera foldery, w których można umieszczać wiadomości. Dlatego też w linii 13 wybierany jest folder domyślny, czyli `Inbox` za pomocą metody `select`. Tutaj również sprawdzany jest kod odpowiedzi.

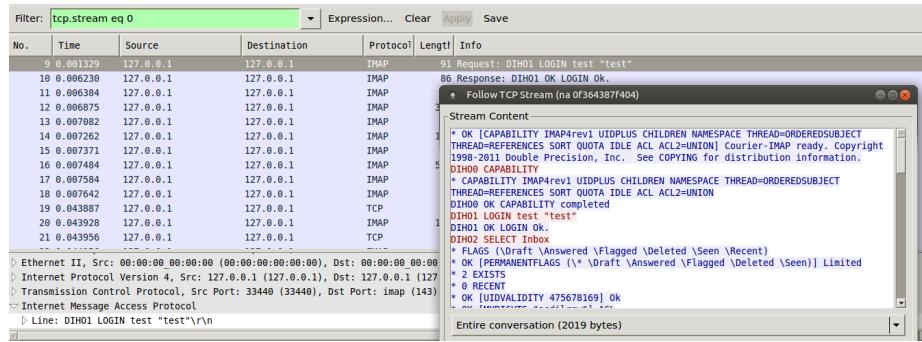
W linii 18, za pomocą metody `search` filtrowane i wybierane są nagłówki wiadomości. Nas interesują wszystkie wiadomości, dlatego drugi parametr ma wartość `ALL`. Pierwszy parametr to wartość pola `CHARSET` przekazanego do serwera, który w tej chwili pewnie nic nie mówi, ale wróćmy do niego podczas pisania skryptu gniazdowego. Wartość `None` oznacza, że pole `CHARSET` nie zostanie w ogóle wysłane. W odpowiedzi otrzymujemy kod oraz listę identyfikatorów (numerów) wiadomości (oddzielonych spacją), które dalej wykorzystamy do pobrania treść wybranej wiadomości.

Polecam wprowadzić w skrypcie wykorzystanie funkcji `print`, żeby podejrzeć wartości poszczególnych zmiennych zwracanych przez serwer.

W końcu, gdy mamy listę identyfikatorów, którą generujemy w linii 20 i

każdy z nich zapisujemy w zmiennej num, możemy wyświetlić wszystkie wiadomości po kolej. Za pomocą metody `fetch` pobieramy wybrane fragmenty wiadomości przekazując do metody numer wiadomości num oraz napis identyfikujący wybrany fragment. W linii 21 pobieramy całą wiadomość przekazując napis (RFC822), zaś w lini poniżej pobieramy jedynie treść wiadomości (bez nagłówków). Po pobraniu wybranych fragmentów, wyświetlamy je na ekranie w liniach 23-27.

Po wyświetleniu wszystkich wiadomości zamykamy skrzynkę `Inbox` za pomocą metody `close` i wylogowujemy się metodą `logout`.



Rysunek 4.6: Pobranie maili (IMAP) - podsłuchanie komunikacji.

Wiemy już jak napisać prostego klienta IMAP, który pobiera wiadomości z domyślnego folderu, wykorzystując bibliotekę `imaplib`. Teraz postaramy się zrobić to samo za pomocą samych gniazd. Znowu musimy poznać protokół oraz składnię i kolejność komend wysyłanych do serwera i odpowiedzi. Ponownie mamy kilka możliwości, żeby te dane uzyskać. Pierwszą z nich jest, jak zwykle, dokumentacja, czyli RFC: <https://tools.ietf.org/html/rfc3501>. Druga, prostsza opcja to podsłuchanie i przeanalizowanie pakietów przesyłanych podczas działania klienta, którego już napisaliśmy. Na Rysunku 4.6 przedstawiam zrzut ekranu z Wiresharka, na której zaznaczyłem jedną z komend, a konkretne komendę `LOGIN`. Oprócz podglądu jednego pakietu możliwe jest skorzystanie z opcji *Follow TCP Stream*, która wyświetla komunikację w ramach jednego połączenia, również przedstawioną na Rysunku 4.6.

Posiadając takie informacje możemy przejść omówienia gniazdowego klienta IMAP, którego kod przedstawiam poniżej.

```

1 import sys
2 import socket
3 import getpass
4
5 def make_tag(nb):
6     return 'CMD' + str(nb).zfill(3)
7
8 def parse_resp(resp, tag):
9     for line in resp.split('\r\n'):
10        if line.startswith(tag):
11            if line.split(' ')[1] == 'OK':
12                return resp

```

```

13     else:
14         print 'Error:', ' '.join(line.split(' ')[2:]),
15         exit(1)
16     print 'Error: Malformed response.'
17     exit(1)
18
19 def parse_search_resp(resp, tag):
20     for line in resp.split('\r\n'):
21         if line.startswith('* SEARCH'):
22             return line.split(' ')[2:]
23         elif line.startswith(tag + ' '):
24             if line.split(' ')[1] != 'OK':
25                 print 'Error:', ' '.join(line.split(' ')[2:]),
26                 exit(1)
27     print 'Error: Malformed response.'
28     exit(1)
29
30 def retrieve_message(sock, tag, msg_id, part):
31     msg = '{} FETCH {} ({})\n'.format(tag, msg_id, part)
32     sock.send(msg)
33     resp = sock.recv(1024)
34     while not resp.split('\r\n')[-2].startswith(tag + ' '):
35         resp += sock.recv(1024)
36     if resp.split('\r\n')[-2].split(' ')[1] != 'OK':
37         print 'Error:', ' '.join(resp.split('\r\n')[-1].split(' ')[2:]),
38         exit(1)
39     if resp.split('\r\n')[-3].startswith('* NO '):
40         print 'Error:', resp.split('\r\n')[-2][5:],
41         exit(1)
42     return '\r\n'.join(resp.split('\r\n')[1:-3])
43
44 if __name__ == "__main__":
45
46     if len(sys.argv) != 3:
47         sys.stderr.write("usage: %s ip port\n", sys.argv[0])
48         exit(1)
49
50     try:
51         addr = sys.argv[1]
52         port = int(sys.argv[2])
53         assert port > 0
54     except:
55         sys.stderr.write("error: invalid port\n")
56         exit(1)
57
58     username = raw_input('Login:')
59     password = getpass.getpass(prompt="Password:")
60
61     sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
62     sock.connect((addr, port))
63
64     sock.recv(1024) # Welcome message
65     CMDINDEX = 1
66
67     tag = make_tag(CMDINDEX)
68     CMDINDEX += 1
69
70     msg = '{} LOGIN {} {}\n'.format(tag, username, password)
71     sock.send(msg)
72     resp = sock.recv(1024)
73     parse_resp(resp, tag)
74
75     tag = make_tag(CMDINDEX)
76     CMDINDEX += 1
77
78     msg = '{} SELECT Inbox\n'.format(tag)
79     sock.send(msg)
80     resp = sock.recv(1024)

```

```

81|     parse_resp(resp,tag)
82|
83|     tag = make_tag(CMDINDEX)
84|     CMDINDEX += 1
85|
86|     msg = '{} SEARCH ALL\n'.format(tag)
87|     sock.send(msg)
88|     resp = sock.recv(1024)
89|     messages_ids = parse_search_resp(resp, tag)
90|
91|     for msg_id in messages_ids:
92|         tag = make_tag(CMDINDEX)
93|         CMDINDEX += 1
94|         print 'Message id:', msg_id
95|         print 'Message body:'
96|         print retrieve_message(sock, tag, msg_id, 'BODY[TEXT]')
97|         tag = make_tag(CMDINDEX)
98|         CMDINDEX += 1
99|         print 'Whole message:'
100|        print retrieve_message(sock, tag, msg_id, 'RFC822')
101|        print '-----'
102|
103|
104|    tag = make_tag(CMDINDEX)
105|    CMDINDEX += 1
106|    msg = '{} CLOSE\n'.format(tag)
107|    sock.send(msg)
108|    sock.recv(1024)
109|
110|    tag = make_tag(CMDINDEX)
111|    CMDINDEX += 1
112|    msg = '{} LOGOUT\n'.format(tag)
113|    sock.send(msg)
114|    sock.recv(1024)
115|
116|    sock.close()

```

Listing 4.41: Klient IMAP

Program rozpoczyna się w linii 44 i do linii 56 weryfikowana jest poprawność argumentów programu. W liniach 59 i 60 skrypt prosi użytkownika o login oraz hasło do serwera pocztowego. W linii 61 tworzone jest gniazdo strumieniowe, następnie gniazdo łączone jest z serwerem podanym w ramach argumentów, po czym w linii 64 pobierana jest wiadomość powitalna z serwera.

W linii 65 ustawiana jest początkowa wartość zmiennej `CMDINDEX`, która będzie wykorzystana do wygenerowania tagu w linii 67 za pomocą funkcji `make_tag`. Każdy tag składa się z napisu `CMD` oraz numeru pobranego ze zmiennej `CMDINDEX`. Po każdym wygenerowaniu tagu wartość `CMDINDEX` jest o jeden zwiększana. Tag to wartość, która jest wysyłana do serwera na początku każdej komendy. Natomiast serwer, w ostatniej linii odpowiedzi, zwraca ten sam tag, po którym występuje kod odpowiedzi i wiadomość. Tag jest wykorzystywany do rozpoznania ostatniej linii odpowiedzi.

W linii 70 przygotowywana jest pierwsza wiadomość, która służy do zalogowania się na serwerze. Składa się ona z tagu, komendy `LOGIN` oraz nazwy użytkownika i hasła. Po wysłaniu wiadomości w linii 71 pobierana jest odpowiedź, która jest następnie sprawdzana za pomocą funkcji `parse_resp`. Funkcja ta, zdefiniowana z linii 8, sprawdza linia po linii odpowiedź z serwera i jeśli znajdzie taką, która rozpoczyna się tagiem, sprawdza kod odpowiedzi, czyli napis wystę-

pujący po tagu. Jeśli napis jest równy OK (oznaczającego pomyślne wykonanie komendy) to zwracana jest treść odpowiedzi z serwera. W przeciwnym razie, gdy kod odpowiedzi nie jest równy OK, skrypt wyświetla informację o błędzie i kończy działanie.

Następnie, w linii 75, generowany jest kolejny tag, by w linii 80 wysłać komendę SELECT Inbox, która wybiera domyślny folder Inbox. Podobnie, jak w przypadku poprzedniej komendy sprawdzana jest odpowiedź serwera. Przy poprawnym wykonaniu komendy powinna rozpoczyna się od tagu i kodu OK, np. CMD0003 OK [READ-WRITE] Ok. Oprócz kodu odpowiedzi serwer zwraca informacje o liczbie wszystkich oraz nowych wiadomości w folderze, a także o flagach, które mogą być przypisane do wiadomości (np. \Seen oznaczająca, że wiadomość została przeczytana).

Kolejna komenda to SEARCH ALL, która jest wysyłana do serwera w linii 87. Komenda ta służy do pobrania identyfikatorów wiadomości w folderze. Argument ALL służy do pobrania wszystkich wiadomości, jednakże można zawieźć tę listę. Na przykład komenda SEARCH NEW zwraca identyfikatory tylko tych, które mają ustawioną flagę oznaczającą, że wiadomość jest nowa, a komenda SEARCH BODY <string> zwraca te wiadomości, w treści których jest fraza <string>.

W linii 89 wykorzystywana jest funkcja parse_search_resp, zdefiniowana w linii 19, do sprawdzenia odpowiedzi na komendę SEARCH. Sprawdza ona wszystkie linie odpowiedzi. Jeśli znajdzie taką, która rozpoczyna się od napisu * SEARCH, to pobiera z niej i zwraca kolejne numery, które identyfikują odnalezione wiadomości. Jeśli natomiast takiej linii nie ma, to sprawdzany jest kod odpowiedzi. Jeśli jest różny od OK to wyświetlany jest komunikat błędu i skrypt kończy działanie.

Po pobraniu identyfikatorów wiadomości, możemy w pętli (linia 91) pobrać treść wszystkich wiadomości po kolei. W tym celu a początku generowany jest kolejny tag, wyświetlany jest identyfikator wiadomości i w linii 96 pobierana jest treść wiadomości za pomocą funkcji retrieve_message, zdefiniowanej w linii 30. Jako argumenty przekazane jest gniazdo, aktualny tak, identyfikator wiadomości oraz nazwa części wiadomości, którą chcemy pobrać (w linii 91 jest to BODY[TEXT]). Funkcja w linii 31 wykorzystuje komendę FETCH, która składa się z tagu, napisu FETCH, identyfikatora wiadomości oraz części wiadomości, ujętej w nawiasy. Część wiadomości może być określona na wiele sposobów, które opisane są w dokumencie RFC. W tym przypadku część BODY[TEXT] odpowiada ona za samą treść wiadomości, bez nagłówków i załączników. Co więcej, można pobrać wiele części wiadomości naraz.

Po wysłaniu komendy FETCH w linii 32 skrypt pobiera odpowiedź i sprawdza kod odpowiedzi. Wcześniej jednak musimy pobrać całą wiadomość, która może mieć więcej niż 1024 znaki, które pobieramy w linii 33. W tym celu sprawdzamy w linii następnej, czy ostatnia pobrana linia rozpoczyna się od tagu wysłanego w komendzie (jak wspomniałem wcześniej, w ten sposób oznaczamy w portokole IMAP koniec wiadomości). Weryfikacja ta odbywa się w linii 34, w której najpierw całą pobraną wiadomość rozbijamy na linie (separatorem \r\n) i sprawdzamy ostatnią linię, czyli tą o indeksie -2 (indeks -1 oznacza

pustą linię, która występuje po znakach `\r\n` kończących wiadomość). Jeśli ostatnia linia nie rozpoczyna się od tagu, to znaczy, że jeszcze nie pobraliśmy całej wiadomości, więc dobieramy kolejne 1024 bajty i sprawdzamy ponownie.

Jak już pobierzemy całą wiadomość to w linii 36 sprawdzamy, czy w ostatniej linii (znowu indeks -2) po tagu jest kod `OK`. Jeśli nie to wyświetlamy błąd i kończymy działanie, bo nie udało się pobrać wiadomości. Jednakże kod `OK` nie musi oznaczać, że nie wystąpił błąd. Na przykład, gdy próbujemy pobrać wiadomość o identyfikatorze, którego nie ma to kod odpowiedzi też będzie równy `OK`. Dlatego w linii 39 sprawdzamy, czy przedostatnia linia nie rozpoczyna się od napisu `* NO`, który oznacza, że nie udało się wykonać komendy, ale błąd nie występuje po stronie protokołu, tylko argumenty przesłane nie pozwalają je poprawne wykonanie komendy. W końcu, jeśli kod odpowiedzi jest `OK` i wiadomość e-mail faktycznie została pobrana, to zwracamy treść odpowiedzi, która również może mieć zmienny format, który zależy od tego, w jaki sposób zdefiniowaliśmy części, które chcemy pobrać. W odpowiedź serwer zwraca (w nawiasach) nazwę każdej z części, po której w nawiasach klamrowych jest liczba znaków, które zajmuje dana część, i po której jest treść tej części. Jeśli poprosiliśmy o więcej niż jedną część to są one wszystkie ujęte w nawiasach okrągłych i umieszczone jedna po drugiej. W naszym przypadku pobieramy tylko jedną część, więc możemy po prostu zwrócić treść odpowiedzi bez pierwszego wiersza (informacja o nazwie części i nawias otwierający) oraz bez ostatnich dwóch wierszy (indeksowanie `[:3]` oznacza od początku do 3. linii od końca), czyli bez nawiasu zamkajającego (przed-przedostatnia linia), kodu odpowiedzi (przedostatnia linia) oraz pustej, ostatniej linii.

Wracamy do programu głównego z treścią odpowiedzi, którą wyświetlamy na ekranie w linii 96. Następnie generujemy nowy tag i ponownie korzystamy z funkcji `retrieve_message`, żeby pobrać i wyświetlić całą wiadomość w linii 100. W tym celu korzystam z nazwy RFC822, która oznacza dokument <https://tools.ietf.org/html/rfc822> opisujący format przesyłanych wiadomości.

Na końcu wysyłamy komendę `CLOSE`, zamkającą folder `Inbox`, następnie komendę `LOGOUT`, wylogowującą użytkownika i zamykamy połączenie.

Dalsze kroki...

- Protokół IMAP obsługuje szyfrowanie TLS i jest ono opisane w dokumencie <https://tools.ietf.org/html/rfc3501>. Zastanów się, w jaki sposób dodać do naszego klienta obsługę TLS.
- Rozbuduj naszego klienta IMAP tak, żeby pobierał sekcję `BODY [TEXT]` i RFC822 w jednym wywołaniu funkcji `retrieve_message`.
- Rozbuduj naszego klienta IMAP tak, żeby po każdym pobraniu i wyświetleniu wiadomości oznaczał ją jako przeczytaną (sprawdź komendę `STORE`).

- Spróbuj napisać serwer poczty, obsługujący protokół IMAP. Nie realizuj faktycznego pobierania e-maili, tylko zasymuluj jego działanie tak, żeby nasz prosty klient IMAP mógł pobrać wiadomości.

4.3 Protokół HTTP

Protokół HTTP to jeden z najpopularniejszych protokołów warstwy aplikacji, który zwykle wykorzystuje protokół TCP z warstwy transportowej. Jednym z założeń protokołu HTTP (przed wersją 2.0) była jego czytelność, czyli miał to być protokół tekstowy, jednakże z możliwością przesyłania danych binarnych.

HTTP to protokół typu zapytanie-odpowiedź. Zapytanie, wysyłane przez klienta, zawiera informację o żądanym zasobie. Odpowiedź, wysyłana przez serwer, zawiera treść zasobu. Jeśli serwer nie jest w stanie zwrócić odpytywanego zasobu, odpowiedź zawiera kod reprezentujący powód, dla którego zasób nie mógł być wysłany (np. zasób nie istnieje).

Najpopularniejszą, wykorzystywaną wersją protokołu HTTP jest wersja 1.1, opisana w dokumentach RFC 2616 oraz od 7230 do 7235. Istnieje jednak nowsza wersja, tj. wersja 2.0, opisana w dokumentach RFC od 7540 i 7541, oparta na protokole SPDY od Google. Jej głównym zadaniem jest przyspieszenie protokołu poprzez wykorzystanie protokołu binarnego, wykorzystanie pojedynczego połączenia TCP, czy mechanizmu PUSH, w którym klient nie musi wysyłać kolejno wiele żądań HTTP, lecz serwer może sam dosyłać zasoby w ramach połączenia.

Protokół HTTP2 nie jest jeszcze powszechnie wspierany przez serwery.

4.3.1 Żądanie HTTP

Przykład żądania HTTP przedstawiono poniżej.

```
GET /nazwa_pliku HTTP/1.1
Host: www.debian.com
User-Agent: Mozilla/5.0 (X11; U; Linux i686; pl; rv:1.8.1.7)
Accept-Language: pl,en-us;q=0.7,en;q=0.3
Accept-Charset: ISO-8859-2,utf-8;q=0.7,*;q=0.7
Connection: close
\r\n
```

W pierwszej linii żądania umieszczono metodę żądania (GET), URI - ścieżkę do zasobu (/nazwa_pliku) oraz wersję protokołu (HTTP 1.1).

Dozwolone metody to:

- GET – pobranie zasobu wskazanego przez URI,
- HEAD – pobiera informacje o zasobie, stosowane do sprawdzania dostępności zasobu,

- PUT – przyjęcie danych przesyłanych od klienta do serwera, najczęściej aby zaktualizować wartość zasobu,
- POST – przyjęcie danych przesyłanych od klienta do serwera (np. wysyłanie zawartości formularzy),
- DELETE – żądanie usunięcia zasobu,
- OPTIONS – informacje o opcjach i wymaganiach dotyczących zasobu,
- TRACE – diagnostyka, analiza kanału komunikacyjnego,
- CONNECT – żądanie przeznaczone dla serwerów pośredniczących pełniących funkcje tunelowania,
- PATCH – aktualizacja części zasobu (np. jednego pola).

Wartość URI (ang. Uniform Resource Identifier) to ścieżka do zasobu na serwerze, która może zawierać dodatkowo parametry HTTP oraz fragment (za znakiem #). Przykład wartości URI z HTTP: /katalog1/katalog2/plik?parametr1=wartosc1&p#

W kolejnych liniach znajdują się nagłówki zapytania. Zostaną one omówione dokładniej w kolejnym podrozdziale. Każdy nagłówek musi być zakończony znakami nowej linii, tj. \r\n.

W zapytaniu HTTP dla wybranych metod (np. PUT, POST) mogą być jeszcze umieszczone dane, które oddzielone są od nagłówków jedną pustą linią. Obecność danych jest sygnalizowana nagłówkiem Content-Length, który zawiera wielkość danych. W przypadku, gdy w zapytaniu nie są przesyłane dane, jest ono zakończone znakiem nowej linii, tj. \r\n.

4.3.2 Odpowiedź HTTP

Przykład odpowiedzi HTTP przedstawiono poniżej.

```
HTTP/1.1 200 OK
Date: Mon, 27 Jul 2009 12:28:53 GMT
Server: Apache
Last-Modified: Wed, 22 Jul 2009 19:15:56 GMT
Accept-Ranges: bytes
Content-Length: 51
Vary: Accept-Encoding
Content-Type: text/plain
\r\n
Hello World! My payload includes a trailing CRLF.
```

W pierwszej linii żądania umieszczono wersję protokołu (HTTP 1.1), kod odpowiedzi (200) oraz odpowiednik tekstowy kodu odpowiedzi (OK). Kody odpowiedzi zostaną omówione dokładnie w dalszym podrozdziale.

W kolejnych liniach znajdują się nagłówki odpowiedzi, które dokładniej zostaną omówione w następnym podrozdziale. Po nagłówkach umieszczana jest pusta linia, czyli znaki `\r\n`.

W dalszej części odpowiedź HTTP może zawierać dane zasobu, jeśli wśród nagłówków istnieje nagłówek Content-Length, który dodatkowo określa wielkość przesyłanych danych.

4.3.3 Nagłówki

Nagłówki może zawierać zarówno zapytanie, jak i odpowiedź HTTP. Składają się one z nazwy oraz wartości oddzielonych dwukropkiem. Każdy nagłówek jest zakończony znakami końca linii, tj. `\r\n`.

Nagłówki wykorzystywane są w różnych celach. Mogą służyć do przesyłania dodatkowych danych do serwera (np. ciasteczka oraz dane uwierzytelniające), do żądania określonego formatu odpowiedzi, czy wielu innych celów.

W protokole HTTP zdefiniowano bardzo dużo nagłówków, dlatego nie ma sensu, aby omawiać je wszystkie. W tym rozdziale omówimy te najpopularniejsze oraz kilka bardziej egzotycznych.

Nagłówek Host

Wielokrotnie na jednym serwerze WWW uruchamiane jest więcej, niż jedna aplikacja. W konsekwencji wszystkie aplikacje posiadają ten sam adres IP. Protokół DNS służy do pobrania tego adresu IP na podstawie domeny serwisu WWW, jednakże łącząc się z pobranym adresem IP serwer nie wie, do której z aplikacji przekazać żądanie HTTP. W protokołach IP, czy TCP, z których HTTP korzysta nie możemy przekazać identyfikatora aplikacji, dlatego jest to określone w nagłówku Host.

Na przykład w sytuacji, gdy na jednym adresie IP (jednej maszynie) uruchomione są dwie aplikacje: aplikacja1.domena.com oraz aplikacja2.domena.com, to gdy chcemy wysłać żądanie do pierwszej aplikacji, umieszczony zostanie nagłówek Host: `aplikacja1.domena.com`.

Nagłówek User-Agent

Nagłówek zawiera informację o kliencie wykorzystywanym przez użytkownika, który wysyła żądanie HTTP. Jako klienta możemy rozumieć przeglądarkę internetową, ale również bibliotekę wysyłającą żądanie, np. curl.

Wartość nagłówka może być wykorzystywana przez serwer do przygotowania odpowiedzi dopasowanej do klienta lub na przykład do generowania statystyk.

Przykład:

```
User-Agent: Mozilla/5.0 (X11; U; Linux i686; pl-PL; rv:1.7.10)
Gecko/20050717 Firefox/1.0.6
```

Nagłówek Connection

Nagłówek pozwala określić klientowi, co serwer powinien zrobić z połączeniem wykorzystanym do przesłania danych. Na przykład wartość `close` oznacza, że połączenie powinno zostać zakończone po wysłaniu odpowiedzi, podczas gdy wartość `keep-alive` oznacza, że połączenie powinno zostać utrzymane. Wtedy w ramach jednego połączenia można wysłać wiele zapytań i odpowiedzi.

Podstawowe nagłówki odpowiedzi

Do podstawowych nagłówków odpowiedzi możemy zaliczyć:

- `Date` - aktualna datę serwera (np. `Date: Tue, 15 Nov 1994 08:12:31 GMT`).
- `Server` - nazwa serwera, czasem zdradzająca jego wersję (np. `Server: Apache/2.4.1 (Unix)`).
- `Expires` - data, do której odpowiedź jest uznana jako niezmieniona (np. `Expires: Thu, 01 Dec 1994 16:00:00 GMT`),
- `Last-Modified` - data ostatniej aktualizacji zasobu (np. `Last-Modified: Tue, 15 Nov 1994 12:45:26 GMT`).

Nagłówki danych

Protokół HTTP może być wykorzystany do przesyłania danych dowolnego typu (np. tekstowych, binarnych, json, itp.). W celu określenia typu oraz kodowania danych wprowadzono nagłówki rozpoczętające się od słowa `Content`. Nagłówki te mogą być umieszczone zarówno w zapytaniu, jak i odpowiedzi HTTP.

Nagłówek `Content-Type` służy do określenia typu danych, zgodnie z typami MIME (ang. Multipurpose Internet Mail Extensions), opisanyymi w RFC 2046. Przykładem nagłówka jest `Content-Type: text/html; charset=ISO-8859-2`, gdzie określono, że przesyłane dane to kod HTML, a zestaw znaków to iso-8859-2, czyli zestaw zawierający polskie znaki.

Nagłówek `Content-Encoding` służy do określenia dodatkowego kodowania, modyfikującego typ danych z nagłówka `Content-Type`. Wartość nagłówka określa, jakiego dekodera należy użyć do odkodowania właściwych danych. Zwykle nagłówek jest wykorzystywany do określenia algorytmu kompresji, aby zmniejszyć rozmiar przesyłanych danych. Przykładem nagłówka jest `Content-Encoding: gzip`, który oznacza, że dane zostały skompresowane algorymem gzip. Inne opcje to `deflate` dla algorytmu kompresji o tej samej nazwie, `compress` dla algorytmu kompresji LZW, `br` dla algorytmu Brotli oraz `identity` dla oznaczenia braku dodatkowego kodowania.

Nagłówek `Content-Language` zawiera język dokumentu zwróconego przez serwer, np. `Content-language: en, pl`. Nie musi on jednak zawierać wszystkich języków, które występują w zwracanych danych.

Nagłówek Content-Length służy do określenia rozmiaru przesyłanych danych. Do danych nie zaliczamy nagłówków. Na przykład, jeśli przesyłany formularz z dwoma polami `username=admin&password=admin1` to nagłówek będzie miał wyglądał następująco `Content-Length: 30`.

Nagłówki negocjacji

Zanim określone zostaną wartości nagłówków danych, klient oraz serwer muszą ustalić, jakie wartości tych nagłówków są w stanie obsłużyć. Na przykład, jeśli klient nie jest w stanie obsłużyć kompresji gzip, to gdy serwer mu prześle odpowiedź z nagłówkiem Content-Encoding: gzip, bedzie ona dla klienta bezużyteczna. W celu negocjacji wartości nagłówków danych służą nagłówki rozpoczętające się od słowa Accept.

Nagłówek Accept służy do określenia akceptowalnych przez klienta typów danych (MIME), a więc wartości nagłówka Content-Type. Przykład nagłówka: `Accept: application/xhtml+xml, application/xml, text/xml; q=0.7, text/html; q=0.5, text/plain; q=0.3`. Jak widać, klient może określić wiele typów, które jest w stanie obsłużyć. Dodatkowo dla każdego typu może zostać określona hierarchia z przedziału 0.0 - 1.0. W przypadku, gdy hierarchia nie zostanie określona, przyjmowana jest wartość 1.0. Zadaniem serwera jest wybranie tych typów, które potrwały zwrócić, po czym ostatecznie wybrany zostanie ten, który ma największy priorytet. Typy danych mogą również zawierać znaki *, które oznaczają dowolną wartość, np. `*/*` lub `text/*`.

Nagłówek Accept-Charset zawiera preferowane przez klienta formaty kodowania. Nagłówek Accept-Charset również może zawierać listę formatów oddzielonych przecinkiem wraz z wartością hierarchii. Na przykład nagłówek `Accept-Charset: utf-8, iso-8859-13; q=0.8` oznacza, że klient oczekuje na kodowanie utf-8, a jeśli serwer nie jest w stanie zwrócić takiego kodowania, to w drugiej kolejności klient oczekuje na kodowanie iso-8859-13.

Nagłówek Accept-Encoding służy do ustalenia wartości nagłówka Content-Encoding, czyli dodatkowego kodowania danych. Analogicznie jak poprzednie nagłówki, w tym również można podać listę obsługiwanych przez klienta kodowań wraz z hierarchią. Na przykład nagłówek `Accept-Encoding: gzip; q=1.0, identity; q=0.5, *; q=0` oznacza, że klient jest w stanie obsłużyć preferowane kodowanie gzip, jak również brak kodowania (identity) i nie akceptuje innych kodowań ($q=0$).

Nagłówki warunkowe

Nagłówki warunkowe pozwalają przesyłać informację do serwera, żeby zwrócił dane tylko wtedy, gdy spełnione są określone warunki. Klasycznym przykładem jest sytuacja, w której przeglądarka cache'uje pobrane wcześniej zasoby (np. pliki CSS, czy JS). Przy następnym żądaniu do tego samego zasobu przeglądarka wyśle do serwera informację, żeby przesłał treść zasobu tylko wtedy, gdy został zmieniony od daty poprzedniego pobrania. Dzięki takim nagłówkom

zmnieszany jest rozmiar przesyłanych danych. Nagłówki warunkowe rozpoczynają się od słowa `If`.

Nagłówek `If-Modified-Since` pozwala warunkowo wykonać żądaną akcję (np. GET, POST) na zasobie tylko wtedy, gdy jego wartość uległa zmianie od daty będącej wartością nagłówka. Na przykład nagłówek `If-Modified-Since: Sat, 29 Jan 2017 19:43:31 GMT` przy żądaniu GET oznacza, że serwer zwróci dane żadanego zasobu, jeśli jego treść uległa zmianie od godziny 19:43:31 29 stycznia 2017 roku czasu GMT. W przeciwnym razie dane zasobu nie zostaną zwrocone, lecz serwer poinformuje klienta, że zasób się nie zmienił.

Nagłówek `If-Unmodified-Since` jest przeciwnieństwem poprzedniego nagłówka. W tym przypadku akcja zostanie wykonana, jeśli od czasu określonego w wartości nagłówka zasób nie uległa zmianie. Nagłówek jest ignorowany w sytuacji, gdy żądanie bez nagłówka zwróciłoby kod inny niż 2xx lub 412. Analogicznie nagłówek jest ignorowany, gdy format daty jest niepoprawny.

Nagłówki ciasteczek

Ciasteczka (ang. cookies) służą do przechowywania przez serwer informacji po stronie klienta. Klasycznym przykładem wartości przechowywanej w ciasteczku jest identyfikator sesji. Ciasteczka wysyłane są do serwera z każdym żądaniem HTTP, dlatego serwer jest w stanie rozpoznać z jakiej sesji pochodzi żądanie i z jakim zalogowanym lub nie zalogowanym użytkownikiem jest powiązane.

Nagłówek `Set-Cookie` to nagłówek odpowiedzi, który służy do zapamiętania w przeglądarce pary klucz i wartość. W nagłówku może zostać umieszczone wiele takich par. Na przykład jeśli serwer chce po stronie przeglądarki zapamiętać parametr `sessionid` o wartości `e14ukv0kqbvoirg7nkp4dncpk3` oraz parametr `user` o wartości `admin`, zwróci nagłówek `Set-Cookie: sessionid=e14ukv0kqbvoirg7nkp4dncpk3; user=admin`. Oprócz wartości ciasteczek w nagłówku może również zostać przesłana informacja o okresie, przez jaki ciasteczko będzie ważne, domenie, na której będzie ważne oraz inne polecenia dla przeglądarki. Na przykład ciasteczko `Set-Cookie: qwerty=219ffwef9w0f; Domain=somecompany.co.uk; Path=/docs; Expires=Wed, 30 Aug 2019 00:00:00 GMT` to parametr o nazwie `qwerty` i wartości `219ffwef9w0f`, który zostanie wysłany przez przeglądarkę tylko w żądaniu, które zostanie przesłane na adres domeny `somecompany.co.uk` i zawiera w ścieżce do zasobu ciąg `/docs`. Dodatkowo, przeglądarka nie wyśle tego ciasteczka po północy 30 sierpnia 2019.

Nagłówek `Cookie` to nagłówek żądania, w którym przeglądarka przesyła parametry i wartości ciasteczka, jeśli zostały spełnione powyższe warunki. Oczywiście wcześniej przeglądarka musi otrzymać żądanie `Set-Cookie` z tymi parametrami i wartościami. Odpowiednikami powyższych nagłówków `Set-Cookie` będą `Cookie: sessionid=e14ukv0kqbvoirg7nkp4dncpk3&user=admin` oraz `Cookie: qwerty=219ffwef9w0f`.

Nagłówki autoryzacji

Protokół HTTP obsługuje autoryzację w ramach nagłówków `WWW-Authenticate` oraz `Authorization`.

Nagłówek `WWW-Authenticate` to nagłówek odpowiedzi HTTP, w którym serwer informuje o metodzie autoryzacji. Nagłówek ten dotyczy tylko tych zasobów, które wymagają autoryzacji, a w żądaniu nie przesłano danych autoryzujących lub przesłane dane są nieprawidłowe. Przykładem nagłówka jest `WWW-Authenticate: Basic realm="XXX"`, który informuje przeglądarkę o tym, że aby dostać się do zasobu należy przesyłać nazwę użytkownika oraz hasło połączone znakiem dwukropka oraz zakodowane algorytmem BASE64. Wartość `realm` (równa `XXX` w przykładzie) to nazwa domeny zasobu. W odpowiedzi na taki nagłówek przeglądarka wyświetli użytkownikowi okno z prośbą o podanie loginu i hasła oraz umieści w tym oknie nazwę domeny. Inna opcja wartości nagłówka to `Digest`, która wymaga przesłania skrótu będącego sekretem znanym tylko serwerowi oraz klientowi. Więcej informacji na temat różnych typów autoryzacji można znaleźć w dokumencie RFC 2617.

Nagłówek `Authorization` to nagłówek żądania, w którym przesyłane są dane autoryzujące zgodnie z algorytmem określonym w nagłówkach `WWW-Authenticate`. Oprócz danych autoryzujących, w nagłówku jest również przesyłana metoda autoryzacji. Jeśli metodą autoryzacji jest `Basic` to w nagłówku przesyłane są nazwa użytkownika oraz hasło połączone znakiem dwukropka oraz zakodowane algorytmem BASE64, tj. `Authorization: Basic QWxhZGRpbjpvcGVuIHNlc2FtZQ==`.

Ciekawe, mniej popularne nagłówki

`Accept-Ranges`, `Content-Range` i `Range`

Zestaw tych nagłówków pozwala zminimalizować wolumen przesyłanych danych i jest szczególnie przydatny, gdy serwer udostępnia duże pliki. Idea polega na pobieraniu części zasobu, zamiast jego całości. Proces rozpoczyna serwer za pomocą jednego z nagłówków negocjacji, tj. `Accept-Ranges`, który umieszcza w nagłówkach odpowiedzi. Standardowymi wartościami nagłówka są `none` oraz `bytes`. Pierwsza opcja oznacza, że serwer nie obsługuje częściowego zwracania zasobu, natomiast druga oznacza, że jeśli żądanie określi, które bajty chce uzyskać (zakres), to serwer zwróci tylko je.

Przykład:

```
GET / HTTP/1.1
Host: www.debian.com
User-Agent: Mozilla/5.0 (X11; U; Linux i686; pl; rv:1.8.1.7)
Connection: close
\r\n

HTTP/1.1 200 OK
Server: Apache
Accept-Ranges: bytes
Content-Length: 1477
```

```
Content-Type: text/plain
\r\n
<HTML>
```

Zgodnie z powyższym przykładem serwer zwrócił informację, że obsługuje częściowe przesyłanie danych. W związku z tym, klient może później odpytując o duży zasób poprosić o jego część, np. pierwsze 1000 bajtów. W tym celu korzysta z nagłówka Range, którego wartość jest równa bytes=<pierwszy bajt>-<ostatni bajt>. W odpowiedzi otrzyma część zasobu od bajtu o numerze <pierwszy bajt> włącznie do bajtu o numerze <ostatni bajt> włącznie. Numerowanie bajtów zaczyna się od zera. Określając zakres możemy pominąć jedną z wartości, np. bytes=<pierwszy bajt>-, czyli żądanie o zasób od bajtu <pierwszy bajt> do końca zasobu lub bytes=-<ostatni bajt>, czyli żądanie o zasób od jego początku do bajtu <ostatni bajt> włącznie. Ponadto, nagłówek może zawierać kilka zakresów, np. jeśli chcemy pobrać pierwsze sto bajtów i ostatnie sto bajtów zasobu wyślemy nagłówek Ranges: bytes=0-100,-100-.

Przykład:

```
GET /duzy_zasob HTTP/1.1
Host: www.debian.com
User-Agent: Mozilla/5.0 (X11; U; Linux i686; pl; rv:1.8.1.7)
Range: bytes=0-1000
Connection: close
\r\n
```

W odpowiedzi na takie zapytanie serwer zwróci odpowiedź z informacją, że jest to część zasobu. Dodatko, jako że protokół HTTP jest bezstanowy, odpowiedź zawiera informację o zwróconym zakresie oraz rozmiarze zasobu (Content-Range: bytes 0-1000/47022).

```
HTTP/1.1 206 Partial content
Server: Apache
Content-Range: bytes 0-1000/47022
Content-Length: 1001
Content-Type: image/gif
\r\n
<DANE>
```

W sytuacji, gdy żądanie zawiera wiele zakresów zasobu, np. Range: bytes=500-999, 7999-, w odpowiedzi umieszczone zostaną wszystkie zakresy w oddzielnych sekcjach. Wtedy w nagłówku odpowiedzi Content-type umieszczona jest informacja, że zwrócony zasób to kilka zakreów danych (multipart/byteranges). Oprócz tego, w atrybutie boundary umieszczony jest losowy napis, który służy do oddzielenia poszczególnych sekcji. Analogiczny mechanizm jest wykorzystywany, gdy przez formularz przesyłane są dane binarne. Ostatecznie nagłówek ma wartość multipart/byteranges; boundary=THIS_STRING_SEPARATES, a zwrócone dane są rozdzielone jak na przykładzie poniżej.

```
HTTP/1.1 206 Partial Content
Date: Wed, 15 Nov 1995 06:25:24 GMT
Last-Modified: Wed, 15 Nov 1995 04:58:08 GMT
Content-type: multipart/byteranges; boundary=THIS_STRING_SEPARATES

--THIS_STRING_SEPARATES
Content-type: application/pdf
Content-range: bytes 500-999/8000

<Dane z 1 zakresu>
--THIS_STRING_SEPARATES
Content-type: application/pdf
Content-range: bytes 7000-7999/8000

<Dane z 2 zakresu>
--THIS_STRING_SEPARATES--
```

If-None-Match, If-Match i ETag

Wcześniej omówiliśmy nagłówki warunkowe, takie jak `If-Modified-Since`, którego zadaniem jest przekazanie informacji serwerowi, żeby nie zwracał zasobu, jeśli nie zmienił się od określonej daty, bo mamy ten zasób zapisany lokalnie. Nagłówki `If-Match` i `ETag` pozwalają na bardziej elastyczne zarządzanie cachem. Nagłówek `ETag` to nagłówek odpowiedzi, w którym umieszczona jest wartość jednoznacznie identyfikująca aktualną treść zasobu (np. wynik funkcji hashującej). Klient odpytując o zasób otrzymuje w nagłówku wartość `ETag`, który zapisuje lokalnie przypisany do zasobu. Następnie, gdy odpytuje ponownie o tem sam zasób, może umieścić zapisaną wcześniej wartość nagłówka `ETag` w nagłówku zapytanie `If-None-Match`. Zadaniem serwera jest weryfikacja, czy aktualny `ETag` zasobu jest zgodny z tym, który został przesłany. Jeśli tak, to zasób nie zostaje zwrócony, a serwer zwraca informację, że nie ma potrzeby pobierać zasób ponownie.

Przykład:

```
GET / HTTP/1.1
Host: www.debian.com
User-Agent: Mozilla/5.0 (X11; U; Linux i686; pl; rv:1.8.1.7)
Connection: close
\r\n

HTTP/1.1 200 OK
Server: Apache
ETag: "6d82cbb050ddc7fa9cbb659014546e59"
Content-Length: 1477
Content-Type: text/plain
\r\n
<HTML>
```

```

GET / HTTP/1.1
Host: www.debian.com
User-Agent: Mozilla/5.0 (X11; U; Linux i686; pl; rv:1.8.1.7)
ETag: "6d82cbb050ddc7fa9cbb659014546e59"
Connection: close
\r\n

HTTP/1.1 304 Not Modified
ETag: "6d82cbb050ddc7fa9cbb659014546e59"
Content-Length: 0

```

Nagłówek `If-Match` działa odwrotnie, czyli akcja jest wykonywana (np. pobranie zasobu z serwera) tylko wtedy, gdy aktualny ETag zasobu na serwerze zgadza się z tym przesłanym w żądaniu.

4.3.4 Kody odpowiedzi

W odpowiedzi na każde żądanie HTTP drugą informacją (po nazwie i wersji protokołu) jest kod odpowiedzi, który informuje klienta, w jaki sposób żądanie zostało lub nie zostało obsłużone. Kody odpowiedzi to liczby trzycyfrowe, gdzie pierwsza z nich określa grupę odpowiedzi.

Pierwsza grupa (1xx) to grupa informacyjna. Wśród kodów informacyjnych popularne to 100 Continue, który oznacza, że serwer przyjął część zapytania i oczekuje na jego dalsze fragmenty. Ten kod jest zwracany, gdy w nagłówku zapytanie będzie nagłówek `Except: 100-continue`. Drugim kodem informacyjnym jest 101 Switching Protocols, który oznacza, że serwer prawnie obsłużył nagłówek Upgrade z zapytania i zmienił wykorzystywany protokół. Kod ten pojawi się później podczas omawiania websocketów.

Druga grupa (2xx) to najpopularniejsza grupa, która oznacza poprawne obsłużenie żądania. W niej są takie kody jak:

- 200 OK oznacza poprawne i pełne wykonanie żądania, zaś treść danych zależy od wykorzystanej metody (GET, POST, itp.).
- 201 Created oznacza, że w ramach obsługi żądania stworzono co najmniej jeden zasób.
- 202 Accepted oznacza, że żądanie zostało przyjęte do zredukowania, ale nie zostało zakończone. Przypadkiem, gdzie ten kod może wystąpić jest zgłoszenie żądania asynchronicznego, zwykle bardzo złożonego i czasochłonnego. Wtedy klient nie oczekuje na zakończenie obsługi, tylko dostaje informację, że żądanie zostanie obsłużone.
- 203 Non-Authoritative Information oznacza, że żądanie zostało obsłużone, ale odpowiedź serwera (200) mogła zostać zmodyfikowana przez proxy. W ten sposób proxy może zasygnalizować klientowi zmianę.
- 204 No Content oznacza, że obsłужenie żądania nie wymaga zwracania żadnych danych, dlatego odpowiedź zawiera tylko nagłówki.

- 205 Reset Content to informacja dla klienta, że po poprawnym obsłużeniu żądania przez serwer powinien on odświeżyć dane widoku, czyli np. odświeżyć stronę.
- 206 Partial Content oznacza, że zwrócone dane to tylko część zasobu. O tym kodzie wspomniałem przy okazji omawiania nagłówka Range.

Grupa trzecia (3xx) oznacza przekierowania, czyli powiadomienie klienta, że zasób znajduje się pod innym adresem. Kody w tej grupie to:

- 300 Multiple Choices oznacza, że dany zasób posiada wiele reprezentacji, czyli jest dostępny pod wieloma innymi adresami, które zwykle są umieszczone w odpowiedzi serwera.
- 301 Moved Permanently oznacza, że zasób nie jest i już nie będzie dostępny pod tym adresem. W nagłówku odpowiedzi Location znajduje się aktualny adres zasobu.
- 302 Found oznacza, że zasób chwilowo jest dostępny pod innym adresem (określonym w nagłówku Location). W przeciwieństwie do kodu 301, klient w przyszłości powinien używać oryginalnego adresu zasobu, a nie jego zmiany.
- 303 See Other oznacza, że po obsłużeniu żądania serwer nie posiada danych do zwrócenia, ale inny zasób (określony w nagłówku Location) opisuje wynik. Na przykład po poprawnym obsłużeniu formularza serwer może przekierować klienta do zasobu, który prezentuje dodany zasób.
- 304 Not Modified oznacza, że nagłówki warunkowe nie zostały spełnione, a zatem zasób nie zmienił swojej treści. Dane zasobu nie są zwarcane.
- 305 Use Proxy to kod uznany jako przestarzały.
- 306 to kod już nie wykorzystywany.
- 307 Temporary Redirect oznacza, że zasób znajduje się tymczasowo pod innym adresem (określonym w nagłówku Location), a klient powinien wysłać żądanie pod nowy adres, ale nie może zmienić metody żądania (GET, POST, itp.).

Grupa czwarta (4xx) zawiera kody oznaczające błąd w protokole, występujący po stronie klienta. Kody z tej grupy to:

- 400 Bad Request oznacza, że serwer nie potrafi przeanalizować zapytania, na przykład ze względu na błędy w składni zapytania.
- 401 Unauthorized oznacza, że żądanie zostało wstrzymane, ponieważ klient go nie autoryzował. W nagłówku odpowiedzi WWW-Authenticate serwer musi zwrócić informację o możliwych formach autoryzacji. Wtedy klient wykorzystuje jedną z nich i dane autoryzujące przesyła w nagłówku Authorization.

- 402 Payment Required to kod zarezerwowany do późniejszego wykorzystania.
- 403 Forbidden oznacza, że dostęp do tego zasobu jest zabroniony. W przeciwnieństwie do kody 401 serwer nie musi zwracać nagłówka, w którym jest informacja, jak uzyskać dostęp. Klient może uwierzytelnić się w pewien (nieokreślony przez standard) sposób, żeby mieć dostęp do zasobu. Na przykład skorzystać z formularza logowania, zalogować się i następnie przesyłać ciasteczkę zawierającą identyfikator autoryzowanej sesji.
- 404 Not Found oznacza, że zasób nie został odnaleziony.
- 405 Method Not Allowed oznacza, że dla danego zasobu wykorzystana metoda (GET, POST) nie jest dozwolona. Na przykład zasoby tylko do odczytu mogą obsługiwać metodę GET, ale nie POST, PUT, czy DELETE.
- 406 Not Acceptable oznacza, że zasób istnieje, ale nie w takiej formie, w jakiej żąda jej klient (nagłówki Accept), więc nie może zostać zwrócony.
- 407 Proxy Authentication Required to kod analogiczny do 401, przy czym dotyczy autoryzacji na poziomie proxy, a nie serwera aplikacji.
- 408 Request Timeout oznacza, że czas obsługi żądania minął. Serwer po zwróceniu tego kodu powinien zamknąć połączenie.
- 409 Conflict oznacza, że podczas próby obsłużenia żądania wystąpił konflikt. Zwykle kod występuje w połączeniu z taką metodą jak PUT. Na przykład, gdy serwer wersjonuje zasoby i jeden klient chce zaktualizować pewien zasób, ale chwilę wcześniej zrobił to inny klient, to żądanie nie może zostać zrealizowane, ponieważ klient chce zmodyfikować nieaktualny zasób.
- 410 Gone oznacza, że dany zasób był, ale już nie jest dostępny. W sytuacji, gdy serwer jest pewien, że ta sytuacja się nie zmieni, to powinien zwracać kod 404. Kod 410 może być wykorzystany w sytuacji, gdy serwer chce, aby wszystkie odnośniki do danego zasobu zostały usunięte. Wtedy, gdy ktoś odpytuje o zasób to wie, że został on celowo usunięty i może usunąć do niego odnośnik.
- 411 Length Required oznacza, że serwer odrzucił żądanie, ponieważ nie zawiera ono nagłówka Content-Length, czyli informacji o rozmiarze przesyłanych danych.
- 412 Precondition Failed dotyczy zapytań zmieniających zasób i oznacza, że warunki z nagłówków zapytania nie zostały spełnione. Na przykład zapytanie modyfikujące PUT z warukiem If-Match zwróci kod 412, jeśli aktualny ETag zasobu nie jest zgodny z tym z nagłówka.

- 413 Payload Too Large oznacza, że rozmiar danych zapytania jest za duży i serwer odmawia jego obsłużenia.
- 414 URI Too Long oznacza, że długość ścieżki do zasobu jest za duża i serwer odmawia obsłużenia zapytania.
- 415 Unsupported Media Type oznacza, że serwer odmawia obsługi żądania, ponieważ format danych nie jest obsługiwany dla danej metody w przypadku żądanego zasobu.
- 416 Range Not Satisfiable oznacza, że serwer odmawia obsługi żądania ze względu na niepoprawne zakresy w nagłówku Range. Może to wystąpić w sytuacji, gdy klient żąda wielu niewielkich, nachodzących na siebie zakresów.
- 417 Expectation Failed oznacza, że serwer nie mógł spełnić żądania z nagłówka Except, czyli zwrócić oczekiwane kodu.
- 426 Upgrade Required oznacza, że serwer odmawia obsługi żądania w ramach aktualnego protokołu, ale informuje klienta o protokole, którego można pomyślnie użyć. Informacja ta zwracana jest w nagłówku Upgrade.

Ostatnia grupa (5xx) zawiera kody oznaczające błąd po stronie serwera. Należą do nich:

- 500 Internal Server Error oznacza, że wystąpił nieoczekiwany, wewnętrzny błąd serwera, który uniemożliwia obsłuszenie żądania.
- 501 Not Implemented oznacza, że serwer nie obsługuje danej metody dla wybranego zasobu.
- 502 Bad Gateway oznacza, że serwer pełniący rolę proxy otrzymał niepoprawną odpowiedź HTTP z właściwego serwera.
- 503 Service Unavailable oznacza, że zasób jest chwilowo niedostępny, np. ze względu na wyczerpanie zasobów lub prace "konserwacyjne".
- 504 Gateway Timeout oznacza, że serwer pełniący rolę proxy nie otrzymał odpowiedzi HTTP z właściwego serwera w dozwolonym czasie.
- 505 HTTP Version Not Supported oznacza, że serwer nie jest w stanie lub odmawia obsługi żądania wykorzystującego daną wersję protokołu HTTP.

4.3.5 Prosty klient HTTP

Celem tego podrozdziału jest napisanie prostego klienta HTTP. Najpopularniejszym klientem HTTP, jaki znamy są oczywiście przeglądarki internetowe, jednakże ich żłożoność jest bardzo duża. Prostszymi klientami HTTP są takie programy jak `curl`, czy `wget`.

Chcąc napisać prostego klienta HTTP musimy odpowiedzieć sobie na kilka pytań, dzięki którym uprościmy nasz program. Główne pytanie to takie, co nasz klient ma zrobić z otrzymaną odpowiedzią HTTP. Przyjmijmy, że nasz klient będzie realizował następujące funkcjonalności:

- będzie pobierał metodą GET zasób podany jako pierwszy argument,
- będzie obsługiwał kody przekierować i automatycznie będzie odpytywał adres podany w przekierowaniu,
- będzie zapisywał dane do pliku o nazwie zgodnej z nazwą zasobu (za ostatnim znakiem `/`) lub pliku `index.html`, jeśli ścieżka do zasobu jest pusta.

W większości języków istnieją biblioteki obsługujące protokół HTTP. Na przykład w Pythonie istnieją biblioteki `urllib` i `urllib2`, które służą do komunikacji z serwerami HTTP. Na nich oparta jest wygodna i prosta w użyciu biblioteka `requests`, dzięki której w prosty sposób możemy pobrać zasób:

```
import requests
res = requests.get('http://domain.com/resource')
print res.text
```

Znając założenia naszego klienta mogę zaprezentować jego kod, który znajduje się na Listingu 4.42.

```
1 import sys
2 import socket
3 import urlparse
4
5 CRLF = "\r\n"
6
7 def recvuntil(s, needle):
8     data = ""
9     while data[-len(needle):] != needle:
10         data += s.recv(1)
11     return data
12
13 def recvall(s, n):
14     data = ""
15     while len(data) < n:
16         data += s.recv(1)
17     return data
18
19 def parse_headers(headers_string):
20     firstline_elems = [
21         s.strip() for s in headers_string.split(CRLF)[0].split(' ')
22     ]
23     protocol, status, status_name = firstline_elems[0], firstline_elems[1], ''.join(
24         firstline_elems[2:])
25     status = int(status)
```

```

25     headers = dict([(s.split(':')[0].lower(), ':' . join(s.split(':')[1:]))]
26         for s in headers_string.strip().split(CRLF)[1:])
27     return status, headers
28
29 def get(url):
30
31     url = urlparse.urlparse(url)
32     path = url.path
33     if path == "":
34         path = "/"
35     HOST = url.netloc if url.port is None else ':' . join(url.netloc.split(':')[1:])
36     PORT = 80 if url.port is None else int(url.port)
37
38     s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
39     s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
40     s.connect((HOST, PORT))
41     s.settimeout(0.3)
42
43     request = "" +
44         "GET %s HTTP/1.1%s" % (path, CRLF) + \
45         "Host: %s%s" % (HOST, CRLF) + \
46         CRLF
47
48     s.send(request)
49     resp_headers = recvuntil(s, CRLF + CRLF)
50
51     status, headers = parse_headers(resp_headers)
52
53     redirected_to = None
54     if 300 <= status < 400:
55         if 'location' in headers:
56             redirected_to = headers['location']
57
58     if redirected_to is None:
59         if 'content-length' in headers:
60             size = int(headers['content-length'])
61             data = recvall(s, size)
62         else:
63             data = ""
64         s.shutdown(1)
65         s.close()
66     return status, data
67
68     s.shutdown(1)
69     s.close()
70     return get(redirected_to)
71
72 def get_filename(url):
73     url = urlparse.urlparse(url)
74     path = url.path
75     if path == "":
76         return 'index.html'
77     return path.split('/')[-1]
78
79 if __name__ == "__main__":
80
81     if len(sys.argv) != 2:
82         sys.stderr.write("usage: %s url\n" % sys.argv[0])
83         exit(1)
84
85     url = sys.argv[1]
86
87     status, data = get(url)
88
89     print 'Server returned status', status
90     if len(data) > 0:
91         filename = get_filename(url)
92         with open(filename, 'wb') as f:

```

```

93     f.write(data)
94     print 'Data saved to file', filename

```

Listing 4.42: Klient HTTP

Właściwy program rozpoczyna się w linii 79. W liniach 81-83 sprawdzamy, czy program otrzymał wymagany argument. Dalsza część programu to wywołanie funkcji `get`, która przyjmuje adres URL jako parametr i zwraca kod odpowiedzi HTTP oraz dane zasobu. Kod odpowiedzi wyświetlony jest na ekranie, a dane zapisywane są w pliku, którego nazwa jest zwrócona przez funkcję `get_filename`.

Funkcja `get` rozpoczyna się w linii 29. Na początku za pomocą funkcji `urlparse` z modułu `urlparse` rozdzielany jest adres URL na poszczególne części, takie jak host, port, czy ścieżka do zasobu (<https://docs.python.org/2/library/urlparse.html>). Do zmiennej `HOST` zapisujemy adres hosta, ale bez portu, jeśli jest on podany w adresie URL. Robimy to w ten sposób, że dzielimy adres po znaku ":"(`split`), z tak powstałej listy zostawiamy wszystkie elementy oprócz ostatniego (`[:-1]`), po czym łączymy te elementy z powrotem znakiem ":"(`join`).

Następnie tworzymy gniazdo strumieniowe i łączymy się z serwerem (38-41) oraz przygotowujemy zapytanie HTTP. W zapytaniu musi wystąpić metoda (GET), ścieżka, którą pobieramy ze sparsowanego adresu URL oraz protokół, który ustaliliśmy na HTTP/1.1. Oprócz tego przesyłamy nagłówek `Host` na wszelki wypadek, gdyby na serwerze było hostowanych wiele domen. Po każdej linii umieszczałyśmy znaki `\r\n`, a za ostatnią z nich dodajemy jeszcze raz te znaki (46).

Po przygotowaniu zapytania wysyłamy je w linii 48, po czym w linii następnej pobieramy część odpowiedzi HTTP zawierającą nagłówki odpowiedzi. Realizujemy to za pomocą funkcji `recvuntil` (zdefiniowanej w liniach 7-11), która jako parametry przyjmuje gniazdo oraz ciąg znaków. Funkcja pobiera dane z gniazda do momentu aż natrafi na wspomniany ciąg znaków. W linii 49 pobieramy dane dopóki nie napotkamy znaków `\r\n\r\n`, czyli znaków kończących część nagłówkową odpowiedzi. Wynik zapisujemy w `resp_headers`.

Następnie za pomocą funkcji `parse_headers` (zdefiniowanej w liniach 19-27) wyciągamy kod oraz nagłówki odpowiedzi z pobranych danych. Kod odpowiedzi funkcja wyciąga z pierwszego wiersza odpowiedzi, rozdzielając go znakiem spacji (linie 20-23) oraz rzutując na typ liczbowy (24). W kolejnych dwóch liniach odpowiedź HTTP jest rozdzielana za pomocą znaków `\r\n` i brane są pod uwagę wszystkie wiersze oprócz pierwszego. W każdym z nich znajduje się nagłówek odpowiedzi, składający się z jego nazwy oraz wartości połączonych dwukropkiem. Dlatego każdy z nagłówków rozdzielimy za pomocą dwukropka, pierwszy element bierzemy jako nazwę nagłówka, zaś pozostałe łączymy z powrotem. Tak przygotowaną listę dwójek rzutujemy na słownik.

Wracając do funkcji `get`, kolejnym krokiem jest sprawdzenie, czy zwrócony kod to nie jest kod przekierowania, czyli czy nie jest to liczba rozpoczynająca się od cyfry 3. Jeśli tak, to sprawdzamy, czy w nagłówkach jest nagłówek `Location`, który powinien zawierać aktualny adres zasobu. Jeśli nagłówek

istnieje, to zapisujemy go w zmiennej `redirected_to` oraz przechodzimy do linii 68, gdzie zamkane jest połączenie i zwracany jest wynik rekursywnie wywołanej funkcji `get` z nowym adresem zasobu. Jeśli on również zwróci kod przekierowania, to ponownie zostanie wywołana funkcja `get`.

Natomiast, jeśli serwer nie zwrócił zasobu przekierowania, to program sprawdzi w linii 59, czy wśród nagłówków istnieje nagłówek `Content-Length`, który określa rozmiar przesyłanych danych. Jeśli istnieje, to z gniazda pobierane jest jeszcze tyle bajtów, ile wskazuje nagłówek. Jeśli nagłówek nie istnieje, to dane są puste. Na końcu połączenie jest zamkane i zwracany jest kod odpowiedzi i dane.

Kod źródłowy klienta HTTP jest dostępny w repozytorium w pliku `http/client.py`.

Co dalej?

- Napisz prosty serwer HTTP (możesz pominąć sprawdzanie błędów), który oprócz nagłówków danych (nagłówki negocjacji możesz pominąć) obsłuży dodatkowo nagłówki związane z ETagiem.
- Zapoznaj się z formatem ZIP (<http://gynvael.coldwind.pl/?id=516>). Napisz program, który pozwoli przejrzeć listę plików znajdujących się w archiwum ZIP hostowanym na serwerze WWW bez potrzeby pobierania całego pliku. Skorzystaj z nagłówka `Range`.

4.3.6 API po HTTP

API (ang. Application Programming Interface), czyli interfejs wystawiany przez aplikację do komunikacji z innymi aplikacjami ma bardzo wiele postaci. W ramach wykładu zajmiemy się jedną z architektur tworzenia interfejsu, który wykorzystuje protokół HTTP, cztli REST API.

REST (ang. Representational State Transfer) to architektura korzystająca w pełni z protokołu HTTP, który charakteryzuje się następującymi cechami:

- **Jednorodny interfejs** między klientem i serwerem upraszcza i oddziela architekturę, dzięki czemu każda jej część może ewoluować niezależnie. Cztery główne zasady jednorodznego interfejsu to:
 - Zasoby identyfikowane są za pomocą ścieżek HTTP. Na przykład ścieżka `/books/14` oznacza zasób o nazwie `book` i identyfikatorze równym 14, zaś ścieżka `/books` oznacza zasób będący kolekcją wszystkich książek.

- *Zarządzanie zasobami poprzez reprezentację* oznacza, że po zwróceniu reprezentacji zasobu (np. w formie JSON) wraz z potencjalnymi metadanymi, klient posiada wszystkie informacje potrzebne do modyfikacji lub usunięcia zasobu.
 - *Zrozumiałe komunikaty*. Każdy wysłany lub otrzymany komunikat powinien mieć informacje, które jednoznacznie określają, co należy zrobić z komunikatem (np. jakiego parsera użyć).
 - *Hypermedia as the engine of application state (HATEOAS)* oznacza, że zwrócony przez serwer zasób powinien wśród swoich danych lub metadanych posiadać informację o powiązanych zasobach (np. ich adresy), dzięki czemu klient posiada pełną listę akcji, które można wykonać w ramach zasobu.
- **Bezstanowa komunikacja.** W komunikacji między klientem i serwerem żadna ze stron nie przechowuje konkretu. Wszystkie zapytania i odpowiedzi są od siebie niezależne, co jest zgodne z protokołem HTTP. Wiele aplikacji web korzysta z sesji, które sprawiają wrażenie, że kontekst użytkownika jest zachowywany między żądaniami. Tak naprawdę obsługa sesji polega na tym, że serwer zapamiętuje pewien zestaw danych powiązany z identyfikatorem sesji. Klient natomiast wysyła ten identyfikator w każdym żądaniu w nagłówku z ciasteczkami.
 - **Cache.** Analogicznie do protokołu HTTP klient ma możliwość cacheowania zasobów. W związku z tym odpowiedzi muszą jednoznacznie określić, czy zwrócony zasób może zostać zapamiętany lokalnie, czy nie.
 - **Klient-Serwer** to architektura, z której korzysta REST, analogicznie do protokołu HTTP.
 - **Układ warstwowy** oznacza, że klient nie jest w stanie stwierdzić, czy łączy się bezpośrednio z serwerem, czy z pośrednikiem (proxy). Zadaniem pośredników może być redukcja obciążenia serwerów (ang. load balancing) lub współdzielony cache.
 - **Kod na żądanie (ang. Code-on-Demand)** to cecha opcjonalna, która zakłada, że zwrócony zasób może zawierać program do wykonania, np. w formie Java appletu lub skryptu JavaScript. Opcjonalność tej cechy wynika z faktu, że takie rozwiązanie zaburza czytelność.

REST został skonstruowany tak, żeby wykorzystać jak najwięcej możliwości protokołu HTTP (np. metody i kody), na którym jest oparty. Poniżej przedstawiam tabelę żądań CRUD (Create, Read, Update, Delete) wraz z informacją za pomocą jakich metod oraz kodów dane żądanie jest obsługiwane przez REST API. Żądania mogą być wysyłane do dwóch rodzajów zasobów: kolekcji zasobów (np. użytkownicy) oraz pojedynczego zasobu (np. użytkownik).

Oprócz metod wspomnianych w tabeli można również korzystać z pozostałych metod HTTP, np. z metody OPTIONS, w celu sprawdzenia listy dostępnych metod dla zasobu.

Metoda	CRUD	Kolekcja elementów (np. /users)	Konkrtetny element (np. /users/<id>)
POST	Stwórz (ang. Create)	201 Created z linkiem do zasobu (np. /users/<id>) w nagłówku Location	404 Not Found, 405 Method Not Allowed lub 409 Conflict, jeśli zasób istnieje.
GET	Pobierz (ang. Read)	200 OK z listą zasobów, która dodatkowo może być stronicowana, filtrowana, czy sortowana.	200 OK z danymi konkretnego zasobu.
PUT	Zaktualizuj (ang. Update)	405 Mathod Not Allowed lub 404 Not Found, chyba że API udostępnia możliwość zmiany całej kolekcji.	200 OK lub 204 No Content. Jeśli zasobu o podanym <id> nie ma to 404 Not Found.
PATCH	Zaktualizuj (ang. Update)	405 Mathod Not Allowed lub 404 Not Found, chyba że API udostępnia możliwość aktualizacji fragmentu kolekcji.	200 OK lub 204 No Content. Jeśli zasobu o podanym <id> nie ma to 404 Not Found.
DELETE	Usuń (ang. Delete)	405 Mathod Not Allowed lub 404 Not Found, chyba że API udostępnia możliwość usunięcia całej kolekcji.	200 OK lub 204 No Content. Jeśli zasobu o podanym <id> nie ma to 404 Not Found.

Tablica 4.1: Tabela metod i kodów HTTP wykorzystywanych przez REST

Jak wspomniałem wcześniej REST nie jest standardem, a jedynie stylem architektonicznym API, dlatego wszelkie ustalenia, tajkie jak wspomniana tabela kodów są umowne i mogą być realizowane w różny sposób. Nie należy zatem upierać się przy jednej wersji implementacji.

Jednym z najtrudniejszych zadań podczas tworzenia API, jak również w trakcie programowania, jest nazewnictwo. Nazewnictwo jest istotne, ponieważ jego poprawna realizacja sprawia, że API staje się intuicyjne i łatwe w użyciu. W przypadku REST każdy zasób otrzymuje swój adres URL, przy czym zasobem może być również kolekcja elementów.

Przykładami zasobów występujących w API mogą być:

- użytkownicy systemu,
- kursy, na które użytkownik się zapisał,
- oś czasu postów użytkownika,
- użytkownicy śledzący posty innego użytkownika,
- artykuł o strzelectwie.

Poniżej przedstawiam kilka dobrych praktyk oraz antywzorców dotyczących nazewnictwa zasobów.

Dobre praktyki

- W przeciwieństwie do wielu znanych API, w REST nazewnictwo dotyczy zasobów, dlatego powinno używać się rzeczowników zamiast czasowników (np. /users zamiast /getUsers).
- REST API jest projektowane dla klienta, a nie na podstawie danych. Nie wszystkie dane muszą być w API, a część może mieć zmodyfikowany format (np. posty nie serwowane oddziennie, a w powiązaniu z użytkownikiem /users/<id>/posts).
- Żądanie dodania nowego klienta: `POST http://api.system.com/customers`.
- Żądanie pobrania klienta: `GET http://api.system.com/customers/1356`. Ten sam adres powinien być wykorzystywany do aktualizacji i usunięcia zasobu.
- Żądanie stworzenia nowego zamówienia przez konkretnego klienta. Czy `POST http://api.system.com/orders` będzie dobre? Jest to po-dejście data-centric, a nie client-centric. Skoro aplikacja klienta chce dodać zamówienie dla konkretnego klienta systemu, to lepszą nazwą zasobu byłoby `POST http://api.system.com/customers/<id>/orders`. Metoda GET na tym zasobie miałaby oczywiście zwracać listę zasobów klienta.
- Czy teraz żądanie do pobrania elementów zamówienia powinno mieć adres `GET http://api.system.com/customers/<customer_id>/orders/<order_id>/items` czy `GET http://api.system.com/orders/<order_id>/items`? To

zależy od tego, w jaki sposób aplikacja kliencka będzie korzystać z API, ale nic nie stoi na przeszkodzie, żeby oba adresy były poprawne. Zasoby mogą posiadać kilka adresów.

- Idąc dalej w hierarchii zasobów, żądanie pobrania konkretnego elementu z wybranego zamówienia określonego klienta wyglądałoby tak: GET `http://api.system.com/orders/<order_id>`

Antywzorce

- Ścieżka do zasobu w parametrach zapytania: GET `http://api.system.com/services?op=update_customer/12345`
- Czasownik jako ścieżka zasobu: GET `http://api.system.com/update_customer/12345`
i GET `http://api.system.com/customers/12345/update`.

Przykład - GitHub REST API

Jako przykład REST API wykorzystamy serwis GitHub, którego dokumentacja znajduje się pod adresem `https://developer.github.com/v3/` (aktualna wersja API to v3).

Rysunek 4.7: GitHub - REST API. Źródło: github.com

Wcześniej wspomniałem, że REST to nie jest standard, a jedynie styl pisania API. Dlatego w dokumentacji developerzy GitHuba musieli umieścić wiele dodatkowych informacji, na przykład o formacie zwracanych danych (JSON), o formacie dat i obsłudze stref czasowych, o sposobie autoryzacji (OAuth), czy o obsłudze stronicowania.

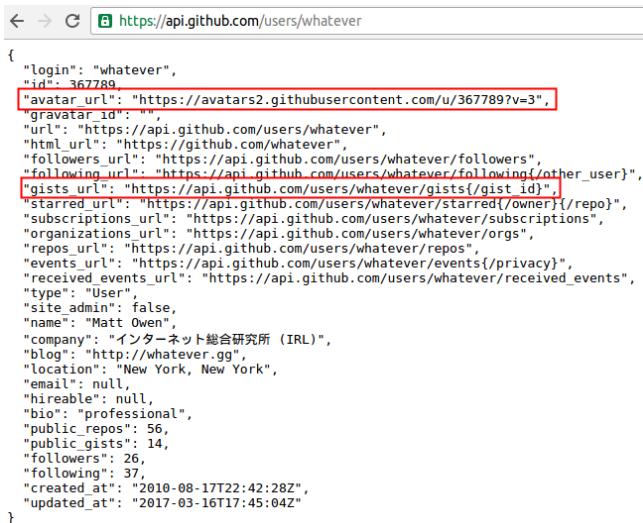
Kilka głównych informacji:

- Aktualna wersja API to v3, a dodatkowo developerzy zachęcają, żeby jednoznacznie informować serwer o chęci wykorzystania wersji v3 za pomocą nagłówka negocjacji, czyli `Accept: application/vnd.github.v3+json`.

- Adres URL API to <https://api.github.com>
- Wszystkie daty są zwracane w formacie ISO 8601, czyli YYYY-MM-DDTHH:MM:SSZ.
- Żądania przyjmują parametry (ang. query parameters), na przykład do filtrowania danych: <https://api.github.com/repos/vmg/redcarpet/issues?state=closed>
- Żądania modyfikujące powinny zawierać dane zakodowane w formacie JSON, z odpowiednim nagłówkiem Content-Type.

W przykładzie skorzystamy z konta użytkownika podanego w parametrze i pobierzemy do pliku jego avatar oraz wszystkie gisty (wklejki). Sprawdźmy najpierw, jak wyglądają wyniki zasobów zwracających poszukiwane dane.

Informacje o użytkowniku znajdziemy w zasobie <https://api.github.com/users/<username>>, którego wynik dla użytkownika whatever znajduje się na Rysunku 4.8.



```
{
  "login": "whatever",
  "id": 367789,
  "avatar_url": "https://avatars2.githubusercontent.com/u/367789?v=3",
  "gravatar_id": "",
  "url": "https://api.github.com/users/whatever",
  "html_url": "https://github.com/whatever",
  "followers_url": "https://api.github.com/users/whatever/followers",
  "following_url": "https://api.github.com/users/whatever/following{/other_user}",
  "gists_url": "https://api.github.com/users/whatever/gists{/gist_id}",
  "starred_url": "https://api.github.com/users/whatever/starred{/owner}{/repo}",
  "subscriptions_url": "https://api.github.com/users/whatever/subscriptions",
  "organizations_url": "https://api.github.com/users/whatever/orgs",
  "repos_url": "https://api.github.com/users/whatever/repos",
  "events_url": "https://api.github.com/users/whatever/events{/privacy}",
  "received_events_url": "https://api.github.com/users/whatever/received_events",
  "type": "User",
  "site_admin": false,
  "name": "Matt Owen",
  "company": "インターネット総合研究所 (IRL)",
  "blog": "http://whatever.gq",
  "location": "New York, New York",
  "email": null,
  "hireable": null,
  "bio": "professional",
  "public_repos": 56,
  "public_gists": 14,
  "followers": 26,
  "following": 37,
  "created_at": "2010-08-17T22:42:28Z",
  "updated_at": "2017-03-16T17:45:04Z"
}
```

Rysunek 4.8: GitHub - Podstawowe informacje o użytkowniku.

Jak widać, w wyniku jest adres do gistów użytkownika, który znajduje się w polu gists_url. Stąd wiemy, że informacje o gistach znajdują się w zasobie <https://api.github.com/users/<username>/gists>, którego wynik dla użytkownika whatever znajduje się na Rysunku 4.9.

Posiadając te informacje, możemy przejść do napisania skryptu, który będzie pobierał avatar i gisty użytkownika do plików. Treść skryptu umieściłem na Listingu 4.43.

Moglibyśmy skorzystać z klienta HTTP, który napisaliśmy wcześniej, jednakże API GitHuba udostępnione jest po protokole HTTPS, czyli szyfrowanym HTTP, a w tej chwili jeszcze nie wiemy jak nazwiać szyfrowane połączenie. Stąd w linii 3 import biblioteki `urllib2` oraz w linii 5 definicja funkcji `get`, która pobiera zasób o adresie przekazanym w parametrze.

```

1 [
2   {
3     "url": "https://api.github.com/gists/e38103708d83801fa5f9",
4     "forks_url": "https://api.github.com/gists/e38103708d83801fa5f9/forks",
5     "commits_url": "https://api.github.com/gists/e38103708d83801fa5f9/commits",
6     "id": "e38103708d83801fa5f9",
7     "git_pull_url": "https://gist.github.com/e38103708d83801fa5f9.git",
8     "git_push_url": "https://gist.github.com/e38103708d83801fa5f9.git",
9     "html_url": "https://gist.github.com/e38103708d83801fa5f9",
10    "files": {
11      "css": {
12        "less": {
13          "filename": "css.less",
14          "type": "text/plain",
15          "language": "less"
16        }
17      }
18    },
19    "public": true,
20    "created_at": "2015-12-02T18:59:53Z",
21    "updated_at": "2015-12-23T20:24:54Z",
22    "description": "",
23    "comments": 0,
24    "user": null
25  }
26]

```

Rysunek 4.9: GitHub - Gisty użytkownika.

W liniach 11-17 głównego programu sprawdzamy, czy podano nazwę użytkownika w parametrze i zapisujemy ją w zmiennej `username`. Następnie w linii 20 tworzymy adres zasobu zawierającego podstawowe dane o użytkowniku i w następnej linii pobieramy wartość zasobu. GitHub zwraca dane w formacie JSON, dlatego w linii 22 korzystam z biblioteki `json` i funkcji `loads`, której przekazuję treść zasobu. Funkcja ta zwraca zdeserializowany obiekt JSON, czyli słownik.

W linii 25 sprawdzamy, czy ustalona jest wartość pola `avatar_url`, które przechowuje adres URL zasobu z avatarem. Jeśli wartość jest ustalona, to w liniach 27-29 pobieramy jego zawartość (funkcja `get`), otwieramy plik o nazwie `<username>.jpg` (w trybie binarnym) i do niego zapisujemy odczytaną treść zasobu.

Analogicznie postępujemy z listą gistów. Najpierw pobieramy zasób zawierający listę (linia 35), a następnie przechodzimy w pętli po każdym giście (linia 39) i pobieramy wszystkie pliki, które dany gist zawiera (pętla w liniach 42-48).

```

1 import sys
2 import json
3 import urllib2
4
5 def get(url):
6     req = urllib2.Request(url=url)
7     return urllib2.urlopen(req)
8
9 if __name__ == '__main__':
10
11     if len(sys.argv) != 2:
12         sys.stderr.write("usage: %s username\n" % sys.argv[0])
13         exit(1)
14
15     APIURL = 'https://api.github.com'
16
17     username = sys.argv[1]
18
19     # Get basic info
20     url = APIURL + '/users/' + username
21     r = get(url)

```

```

22|     basicinfo = json.loads(r.read())
23|
24|     # Download avatar
25|     if basicinfo['avatar_url']:
26|         print 'Downloading avatar...'
27|         r = get(basicinfo['avatar_url'])
28|         with open(username+'.jpg', 'wb') as f:
29|             f.write(r.read())
30|         print 'Downloaded'
31|
32|     # Get gists
33|     print 'Downloading gists...'
34|     url = APIURL + '/users/' + username + '/gists'
35|     r = get(url)
36|     gists = json.loads(r.read())
37|     print ' Found %d gists' % len(gists)
38|
39|     for g in gists:
40|         gid = g['id']
41|         print " Downloading gist %s" % gid
42|         for f in g['files']:
43|             print "     Downloading file %s" % f
44|             url = g['files'][f]['raw_url']
45|             r = get(url)
46|             with open(gid + '_' + f, 'wb') as ff:
47|                 ff.write(r.read())
48|             print "     Downloaded"
49|     print " Downloaded"

```

Listing 4.43: Klient dla GitHuba

Co dalej?

- Napisz klienta REST API, który w parametrach będzie przyjmował metodę, zasób oraz dane na podstawie których wyśle odpowiednie żądanie do serwera udostępniającego REST API i zwróci informację o pomyślnym wykonaniu lub o błędzie.
- Napisz klienta REST API dla GitHub, który będzie przyjmował login i hasło, a następnie korzystając z podanych danych uwierzytelniających odpyta API o dane użytkownika. Zasób zwracający te dane jest dostępny pod adresem: <https://api.github.com>.

4.3.7 Websocket

WebSocket (RFC 6455) jest protokołem opartym na protokole TCP, który umożliwia komunikację dwukierunkową między serwerem i klientem [10]. W przeciwieństwie do HTTP, który jest protokołem żądanie-odpowiedź, WebSocket pozwala na asynchroniczną komunikację. Po zestawieniu połączenia, obie strony mogą wymieniać się danymi w dowolnym momencie, wysyłając pakiet danych. Inną możliwością jest wykorzystanie zapytań asynchronicznych (XHR). W tym przypadku jednak, uzyskanie efektu komunikacji dwukierunkowej z jak najmniejszym opóźnieniem, osiągane jest kosztem zwiększenia ilości zapytań do

serwera. I tak, w związku z zapotrzebowaniem na implementację prawdziwej dwukierunkowej komunikacji w aplikacjach WWW, zaproponowano wdrożenie protokołu WebSocket.

Strona zainteresowana nawiązaniem połączenia, wysyła do serwera żądanie inicjalizujące połączenie (ang. handshake). Żądanie to, ze względów na kompatybilność z serwerami WWW, jest niemal identyczne jak standardowe zapytanie HTTP.

```
GET /chat HTTP/1.1
Host: server.example.com
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Key: dGhIHNhbXBsZSBub25jZQ==
Origin: http://example.com
Sec-WebSocket-Protocol: chat
Sec-WebSocket-Version: 13
\r\n
```

Nagłówek `Upgrade` informuje o chęci zmiany porotokołu na `websocket`. Nagłówki `Sec-WebSocket-Protocol` oraz `Sec-WebSocket-Version` służą do poinformowania serwera, z jakiego protokołu (i w której wersji) klient chce skorzystać. Jest to przydatna opcja, gdy serwer udostępnia kilka protokołów wspieranych przez `websocket`. Nagłówek `Sec-WebSocket-Key` wbrew nazwie nie zawiera klucza, lecz losowy ciąg znaków zakodowanych `base64` w celu usunięcia problemów z pamięcią podręczną (ang. `cache`).

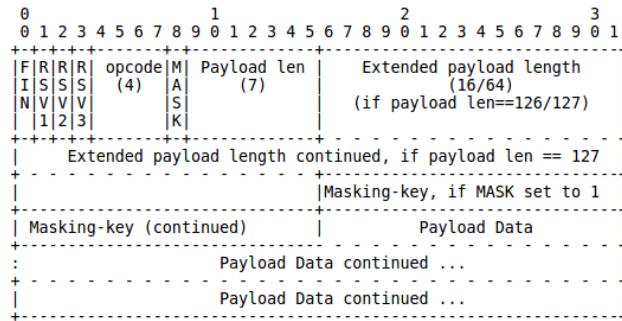
Serwer na takie żądanie wysyła następującą odpowiedź:

```
HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept: HSmrc0sMlYUkAGmm5OPpG2HaGWk=
Sec-WebSocket-Protocol: chat
\r\n
```

Kod odpowiedzi 101 (patrz opis protokołu HTTP wyżej) oznacza, że protokół dla tego połączenia został zmieniony na taki, jaki jest wpisany w nagłówku `Upgrade`, czyli `websocket`. W nagłówku `Sec-WebSocket-Protocol` serwer zwraca nazwę protokołu, który otrzymał w zapytaniu jako potwierdzenie wyboru tego protokołu. Natomiast w nagłówku `Sec-WebSocket-Accept` serwer przesyła zakodowany `base64` wynik funkcji `SHA1` z otrzymanej w żądaniu wartości nagłówka `Sec-WebSocket-Key` połączonej ze stałym GUID równym `"258EAFA5-E914-47DA-95CA-C5AB0DC85B11"`.

Po pomyślnym zakończeniu nawiązywania połączenia, dalsza komunikacja odbywa się poprzez socket TCP już z pominięciem protokołu HTTP. Ramka `WebSocket` umieszczona jest na Rysunku 4.10.

Definicja istotnych pól jest następująca:



Rysunek 4.10: WebSocket - ramka.

- opcode (4 bity) - określa, w jaki sposób należy interpretować ramkę:
 - 0 - dane są kontyfuaracją poprzedniej ramki,
 - 1 - dane w formie tekstowej,
 - 2 - dane w formie binarnej,
 - 3-7 - zarezerwowane,
 - 8 - chęć zakończenia połączenia,
 - 9 - ping,
 - 10 - pong,
 - 11 - 15 - zarezerwowane.
- payload len (7 bitów) - długość danych w ramce (jeśli długość jest mniejsza lub równa 125 bajtów).
- extended payload length (16 bitów) - jeśli wartość poprzedniego pola jest równa 126, to długość danych jest zawarta w tych 16 bitach.
- extended payload length continued (48 bity) - jeśli wartość pola payload len jest równa 127, to długość danych jest zawarta w 8 bajtach z pól extended payload length oraz extended payload length continued.
- MASK (1 bit) - oznaczenie, czy dane są maskowane. Każdy pakiet od klienta do serwera powinien być maskowany. Służy do ochrony cache poisoning.
- masking key (32 bity) - klucz, którym zamaskowano dane (jeśli pole MASK jest ustawione). Maskowanie polega na xorowaniu każdego bajtu danych z kolejnym (powtarzanym cyklicznie) bajtem klucza.

Klient WS

W przykładzie klienta aplikacji skorzystamy z publicznie dostępnego serwera echo, dostępnego pod adresem `ws://echo.websocket.org` (ws to schema dla protokołu WebSocket). Przykład jego wykorzystania jest opisany na stronie <http://websocket.org/echo.html>. Jako, że WebSocket jest wykorzystywany głównie przez serwisy WWW, to bardzo często klienci są pisani w języku Javascript. Na listingu 4.44 umieściłem kod przykładowego klienta WebSocket, pobranego ze strony `ws://echo.websocket.org`.

```
1 <!DOCTYPE html>
2 <meta charset="utf-8" />
3 <title>WebSocket Test</title>
4 <script language="javascript" type="text/javascript">
5 var wsUri = "ws://echo.websocket.org/";
6 var output;
7
8 function init()
9 {
10   output = document.getElementById("output");
11   testWebSocket();
12 }
13
14 function testWebSocket()
15 {
16   websocket = new WebSocket(wsUri);
17   websocket.onopen = function(evt) { onOpen(evt) };
18   websocket.onclose = function(evt) { onClose(evt) };
19   websocket.onmessage = function(evt) { onMessage(evt) };
20   websocket.onerror = function(evt) { onError(evt) };
21 }
22
23 function onOpen(evt)
24 {
25   writeToScreen("CONNECTED");
26   doSend("WebSocket rocks");
27 }
28
29 function onClose(evt)
30 {
31   writeToScreen("DISCONNECTED");
32 }
33
34 function onMessage(evt)
35 {
36   writeToScreen('<span style="color: blue;">RESPONSE: ' + evt.data+'</span>');
37   websocket.close();
38 }
39
40 function onError(evt)
41 {
42   writeToScreen('<span style="color: red;">ERROR:</span> ' + evt.data);
43 }
44
45 function doSend(message)
46 {
47   writeToScreen("SENT: " + message);
48   websocket.send(message);
49 }
50
51 function writeToScreen(message)
52 {
53   var pre = document.createElement("p");
54   pre.style.wordWrap = "break-word";
55   pre.innerHTML = message;
```

```

56     output.appendChild(pre);
57 }
58 window.addEventListener("load", init, false);
59 </script>
60 <h2>WebSocket Test</h2>
61 <div id="output"></div>
62
63
64
65

```

Listing 4.44: Klient WebSocket

Kod źródłowy klienta WS jest dostępny w repozytorium w pliku ws/echo.html.

W kodzie HTML umieszczony jest div o identyfikatorze output, w którym wyświetlane będą informacje o stanie połączenia i dane pobrane przez websocket. Ponadto, na końcu skryptu, w linii 59, funkcja addEventListener powoduje, że po załadowaniu strony zostanie uruchomiona funkcja init z linii 8. W funkcji tej zapisujemy w zmiennej output element div, w którym będziemy wyświetlać informacje oraz wywołujemy funkcję testWebSocket z linii 14.

Funkcja ta tworzy w zmiennej websocket gniazdo (korzystające z protokołu WebSocket oczywiście) i przekazuje mu adres ze zmiennej wsUri (linia 5). Następnie przypisuje własne funkcje do wydarzeń zgłaszanych przez websocket (linie 17-20). Na przykład, gdy połączenie zostanie otwarte, wywołana zostanie funkcja z linii 17, która wywołuje funkcję onOpen z linii 23, przekazując jej obiekt reprezentujący to zdarzenie. Po stworzeniu websocketu próbuje on się połączyć z serwerem.

Po otwarciu połączenia (linia 23) na ekranie wyświetlony zostanie komunikat CONNECTED za pomocą funkcji writeToScreen (linia 51), która po prostu dodaje napis do elementu output wraz ze znakiem nowej linii. Następnie, w linii 26, wysyłane są dane "WebSocket rocks" do serwera za pomocą funkcji doSend (linia 45), która po prostu wyświetla to co wysyła na ekran i za pomocą metody send przesyła dane do serwera. Jako, że serwer to serwer echo, po otrzymaniu napisu odsyła go, co wyzwala wydarzenie onmessage, któremu w linii 19 zostało przypisane wywołanie funkcji onMessage z linii 34. Funkcja ta wyświetla na ekranie otrzymany napis (w kolorze niebieskim) i zamyka połączenie (linia 37). Wynik działania tego klienta umieściłem na Rysunku 4.11.

Na Rysunku 4.12 umieściłem zrzut z Wiresharka, w którym przechwyciłem komunikację między powyższym klientem (uruchomionym w przeglądarce), a serwerem. Jak widać na początku wysyłane są zapytanie i odpowiedź HTTP zgodnie z wcześniej opisanym schematem. Dopiero później wysyłane są dane w protokole WebSocket.

Serwer WS

W celu napisania serwera można skorzystać z szeregu bibliotek: Socket.io (JavaScript), Ratchet (PHP), WebSocketHandler (.NET), Autobahn (Python).

WebSocket Test

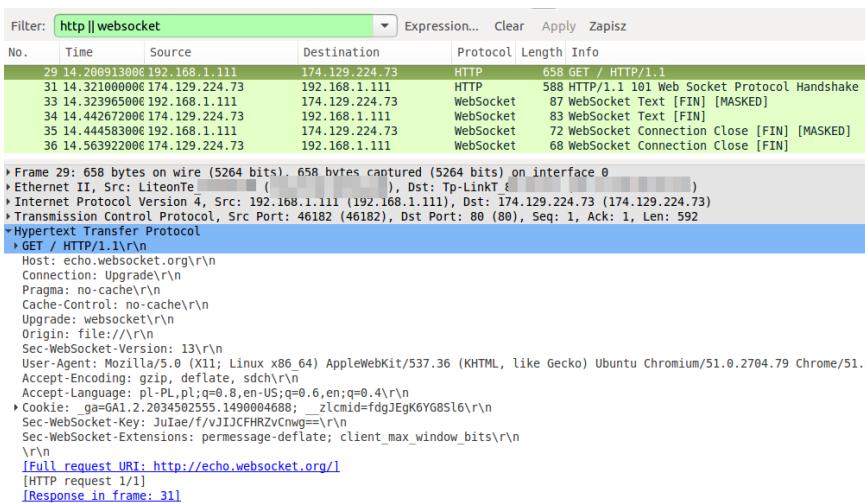
CONNECTED

SENT: WebSocket rocks

RESPONSE: WebSocket rocks

DISCONNECTED

Rysunek 4.11: WebSocket - klient.



Rysunek 4.12: WebSocket - komunikacja.

Można również oczywiście zaimplementować protokół WS samemu korzystając ze zwykłych gniazd.

Co dalej?

- Podejrzyj w Wiresharku komunikację naszego klienta z serwerem echo i napisz własnego klienta, który wykorzystuje zwykłe gniazda do połączenia się w serwerem. W tym celu trzeba najpierw przeprowadzić komunikację w protokole HTTP, a później wysyłać dane po TCP za pomocą protokołu WS.
- Napisz bibliotekę implementującą protokół WebSocket, z której można skorzystać zaimplementowania serwera i zaimplementuj go.

4.4 Analiza protokołu

W tym rozdziale nasze zadanie będzie polegało na przeanalizowaniu i zrozumieniu zasad działania nieznanego nam protokołu. Założymy, że dostajemy zadanie następującej treści:

W domu zainstalowano system czujników i detektorów oraz zintegrowany system zarządzania wszystkimi znajdującymi się w budynku instalacjami. Wszystkie urządzenia komunikują się za pomocą protokołu KNX. Ponadto, w domowej sieci istnieje router KNX, który pozwala na zarządzanie instalacjami z domowej sieci IP. Napisz program, który umożliwi zarządzanie urządzeniami w sieci KNX, np. zapalenie żarówek.

Zadanie zgoła wydaje się proste. Mamy sieć połączonych urządzeń i naszym zadaniem jest napisanie programu, który dostanie się do tej sieci i wyśle komendę do żarówki, dzięki której żarówka się zaświeci. Główną trudnością jest wykorzystanie nieznanego protokołu KNX, ponieważ narzuca go wdrożona instalacja. W ramach rozdziału przeanalizujemy tylko część protokołu, tj. nie będziemy poznawać jego wszystkich możliwości.

Skrypty oraz dokumenty, które są wykorzystywane w tym rozdziale można znaleźć w folderze *knx* w repozytorium. Istnieje tam również skrypt symulujący serwer wystawiony przez KNX. Można go uruchomić z folderu */knx* komendą `python3 knx-server.py 3671 0/0/1`.

Zakładając, że nic nie wiemy o inteligentnych domach i rozwiązaniu od KNX Assoc. zadanie można rozdzielić na następujące korki:

1. Wysokopoziomowe zrozumienie działania urządzeń w sieci KNX.
2. Przeanalizowanie sposobu wystawienia sieci KNX do sieci IP (wspomniany router).
3. Przeanalizowanie ruchu w sieci KNX i inżynieria wstępna protokołu.
4. Napisanie klienta KNX, łączącego się z routerem.

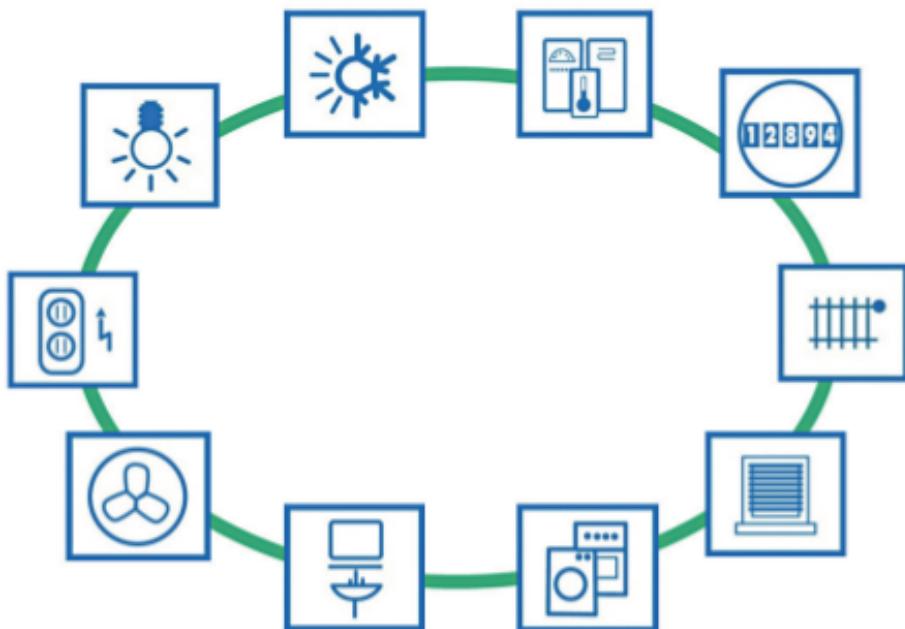
4.4.1 KNX

Pierwszy krok to ogólne zrozumienie zasady działania urządzeń wpiętych w sieć KNX. W tym celu wystarczy skorzystać z wyszukiwarki Google, żeby znaleźć dokumenty i prezentacje omawiające rozwiązania KNX. Nie będzie z tym problemu, ponieważ jest to przemysłowe rozwiązanie i stowarzyszeniu KNX zależy na jego rozpowszechnianiu. W moim przypadku Google na hasło *knx* odpowiedział 8 milionami wyników.

Pierwsze źródło to (oczywiście) Wikipedia (<https://pl.wikipedia.org/wiki/KNX>), z której dowiadujemy się, że KNX umożliwia wspólną komunikację pomiędzy wszystkimi odbiornikami energii elektrycznej w budynku. Opcjonalnie zapewnia

zdalny dostęp do wszystkich instalacji budynkowych i umożliwia dowolne rozwijanie funkcjonalności automatyki budynkowej przez dowolnych producentów z całego świata. Jest wspólnym językiem komunikacji dla wszystkich instalacji budynkowych. Może łączyć instalacje elektryczne, teletechniczne, HVAC, alarmowe, nagłośnienia, monitoring, zabiegów bezpieczeństwa budynkowe, pomiarowanie i wszystkie inne działające w budynku. Wiemy zatem, że jest to standard, z którego korzystają producenci urządzeń. Dzięki temu urządzenia wielu producentów mogą ze sobą współpracować i nie jesteśmy zdani na łaskę tylko jednego producenta.

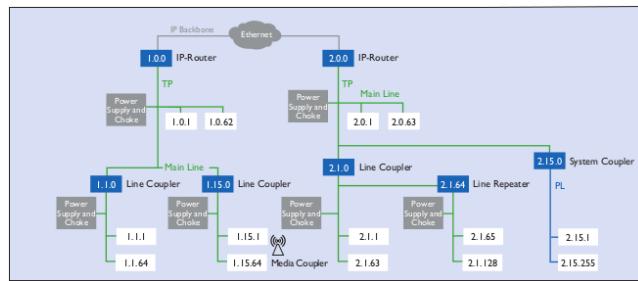
Dowiedzmy się coś więcej o samym standardzie KNX oraz o jego działaniu z oficjalnych dokumentów Stowarzyszenia KNX. Na stronie stowarzyszenia (www.knx.org) jest opublikowany dokument *KNX Basics* (dostępny w repozytorium), który ogólnie opisuje standard i protokół KNX. Zgodnie z nim KNX to system oparty na magistrali (ang. bus), do której podłączone są wszystkie czujniki (ang. sensor), czyli urządzenia monitorujące otoczenie oraz aktorzy (ang. actuator), czyli urządzenia wykonujące akcje w odpowiedzi na monitorowane zmiany (np. podniesienie rolet, gdy w budynku jest ciemno).



Rysunek 4.13: KNX - magistrala. Źródło: Knx.org - KNX Basics

Każde urządzenie podłączone do magistrali posiada swój indywidualny adres składający się z 3 elementów: numer strefy, numer linii oraz numer urządzenia, jak widać na Rysunku 4.14. Urządzenia wymieniają między sobą dane (tzw.

telegramy), w ramach których informują o zmianach w otoczeniu lub wysyłają komendy (np. włączenie lub wyłączenie urządzenia). Jako, że wszystkie urządzenia są podłączone do tej samej magistrali (system zdecentralizowany) to w ramach komunikatów przesyłane są również informacje kontrolne, takie jak np. źródło oraz cel komunikatu.



Rysunek 4.14: KNX - topologia sieci. Źródło: Knx.org - KNX Basics

KNX wspiera 4 różne sposoby wymiany danych między urządzeniami. Pierwszy z nich to KNX Twisted Pair (KNX TP), w którym wszystkie urządzenia połączone są kablem (ang. bus cable), dostarczającym zasilanie oraz przesyłającym dane. Zgodnie z dokumentem KNX Basics jest to najpopularniejsze rozwiązanie.

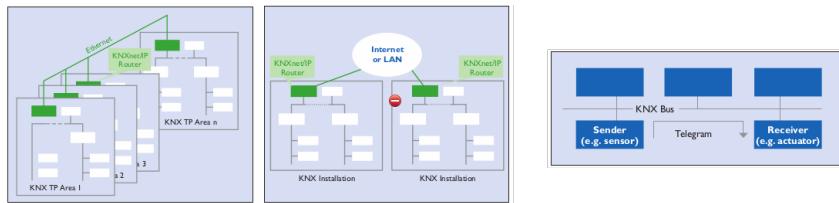
Drugim sposobem jest KNX Powerline (KNX PL), czyli wykorzystanie istniejącej sieci elektrycznej. Rozwiązanie to jest szczególnie przydatne w sytuacji, gdy sieć urządzeń jest instalowana w istniejącym już (np. starym) budynku. Różnicą w stosunku do pierwszego medium jest napięcie, równe 230V w KNX PL oraz 30V dla zasilania i 24V dla przesyłu danych w KNX TP. Ponadto, różni się również prędkością przesyłu danych na korzyść pierwszego sposobu.

Trzeci sposób to KNX Radio Frequency (KNX RF), czyli komunikacja bezprzewodowa. Rozwiązanie idealnie pasujące, gdy nie można ani zainstalować sieci kabli, ani skorzystać z istniejącej sieci elektrycznej.

Ostatni, ale najważniejszy z punktu widzenia niniejszego skryptu sposób to wykorzystanie sieci Ethernet, czyli KNX IP. Najważniejszy, ponieważ w zadaniu jest informacja, że w sieci KNX istnieje router, z którym można połączyć się z lokalnej sieci IP. Co to znaczy? To znaczy, że na przykład ten router KNX jest połączony kablem Ethernet z routerem udostępniającym sieć Wifi, czyli z każdego urządzenia (np. laptopa, komórki) w sieci mamy możliwość połączenia się z routerem KNX.

We wszystkich wspomnianych sposobach format przesyłanych telegramów się różni, jednakże dla nas istotny jest jedynie format telegramu KNX IP, ponieważ właśnie ten sposób komunikacji będziemy obsługiwać. Nie jest dla nas istotne jakiego medium używają pozostałe urządzenia, czyli te "za" routerem KNX, ponieważ my musimy skomunikować się tylko z routerem, a dalsza komunikacja zostanie nawiązana przez niego.

Przykłady takiego rozwiązania znajdują się na Rysunku 4.15, na którym



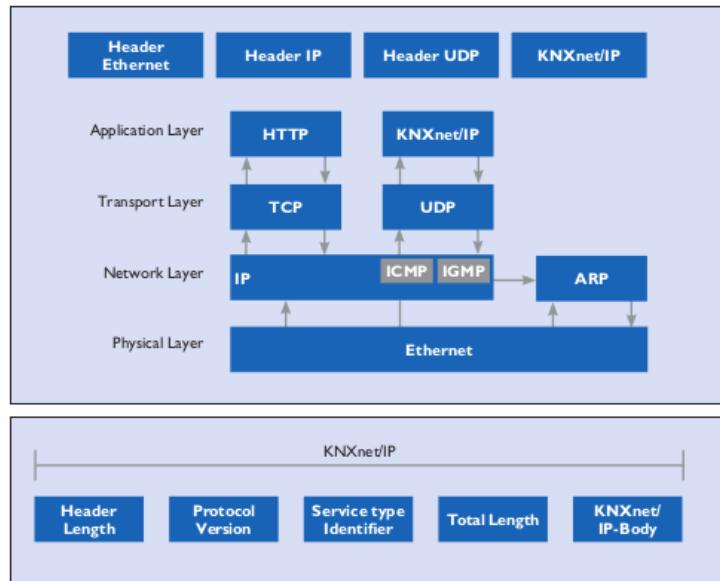
Rysunek 4.15: KNX IP - router i telegram. Źródło: Knx.org - KNX Basics

widać widać, że każda strefa KNX ma własny router KNXnet/IP, który wpięty jest w sieć Ethernet oraz posiada wyjście KNX TP, którym jest zasilany i którego używa do komunikacji z urządzeniami w swojej strefie (ang. area).

Podsumowując, wiemy, że będziemy komunikować się z routerem KNXnet/IP, który przekaże nasze dane do urządzeń w sieci. Pora na poznanie protokołu z którego router korzysta.

4.4.2 Protokół KNXnet/IP

Do komunikacji z urządzeniami KNX wystawionymi do sieci Ethernet (np. wspomniany w zadaniu router) wykorzystuje się protokół KNXnet/IP. Na Rysunku 4.16, w jego górnej części umieszczono go w odniesieniu do innych znanych protokołów na modelu OSI.



Rysunek 4.16: KNXnet/IP - struktura telegramu. Źródło: Knx.org - KNX Basics

Jak widać na modelu OSI KNXnet/IP jest protokołem warstwy aplikacji i korzysta z protokołu UDP. Zgodnie z *KNX Basics* umożliwia 2 tryby działania: tunelowanie (KNXnet/IP Tunelling) i routing (KNXnet/IP Routing). Tunelowanie jest wykorzystywane do przesyłania wiadomości do magistrali z lokalnej sieci Ethernet lub sieci Internet w celu przesyłania telegramów (jak w naszym zadaniu) lub na przykład w celu programowania urządzeń. Routing służy do przesyłania telegramów między dwoma strefami KNX połączonymi siecią Ethernet. Nas oczywiście interesuje tunelowanie.

Strukturę telegramu KNXnet/IP przedstawiono w dolnej części Rysunku 4.16. Więcej informacji o poszczególnych polach można znaleźć w dokumencie *IP Communication* (dostępnym w repozytorium). Telegram składa się z:

- Header Length (1 bajt) - długość nagłówka, która ma zawsze taką samą wartość. Mimo stałej wartości jest ona przesyłana, ponieważ może się zmienić w przypadku wdrożenia nowej wersji protokołu. Jest wykorzystywana do znalezienia potencjalnego początku ramki KNXnet/IP. Wartość: 06_{hex} .
- Protocol Version (1 byte) - wersja wykorzytanego protokołu KNXnet/IP, która aktualnie jest równa 1.0. Wartość: 10_{hex} .
- Service Type Identifier (2 bajty) - identyfikator akcji, która ma być wykonana. Poniższa tabela przedstawia zakresy wartości pola z przypisanymi akcjami. Dokładny spis wszystkich akcji jest dostępny w dokumencie *KNX*

Address range in hex		KNXnet/IP action
From	To	
0200	020F	KNXnet/IP Core
0310	031F	KNXnet/IP Device Management
0420	042F	KNXnet/IP Tunelling
0530	053F	KNXnet/IP Routing
0600	06FF	KNXnet/IP Remote Logging
0740	07FF	KNXnet/IP Remote Configuration and Diagnosis
0800	08FF	KNXnet/IP Object Server

specifications, w wolumenie 3 (System Specifications), części 8 (KNXnet/IP), rozdziale 1. Niestety nie jest to dokument ogólnodostępny, dlatego będziemy musieli szukać tych wartości w innych miejscach.

Poniżej umieściłem listę najczęściej wykorzystywanych akcji, która jest wystarczająca na potrzeby tego rozdziału.

- 0x0201 - SEARCH_REQUEST
- 0x0202 - SEARCH_RESPONSE

- 0x0203 - DESCRIPTION_REQUEST
- 0x0204 - DESCRIPTION_RESPONSE
- 0x0205 - CONNECTION_REQUEST
- 0x0206 - CONNECTION_RESPONSE
- 0x0207 - CONNECTIONSTATE_REQUEST
- 0x0208 - CONNECTIONSTATE_RESPONSE
- 0x0209 - DISCONNECT_REQUEST
- 0x020A - DISCONNECT_RESPONSE
- 0x0420 - TUNNELLING_REQUEST
- 0x0421 - TUNNELLING_ACK
- 0x0310 - DEVICE_CONFIGURATION_REQUEST
- 0x0311 - DEVICE_CONFIGURATION_ACK
- 0x0530 - ROUTING_INDICATION

- Total Length (2 bajty) - całkowity rozmiar ramki KNXnet/IP, wliczając powyższe pola. Jeżeli długość pakietu jest większa niż 252 bajty, pierwszy bajt jest równy FF_{hex} , zaś drugi zawiera pozostałą długość ramki.
- KNXnet/IP Body (zmienna długość) - długość oraz struktura tego pola zależy od identyfikatora typu usługi, wspomnianego wcześniej. To, jaki format ma struktura dla konkretnej wartości usługi będziemy określać w rozdziale dotyczącym analizy ruchu, ponieważ będzie to od nas wymagało wygenerowania lub podsłuchania takiego ruchu. Oczywiście istnieją dokumenty, które to opisują, ale w ramach analizy ruchu poznamy nowe źródło informacji.

Tunelowanie

We wcześniejszej części tego rozdziału wspomniałem, że protokół KNXnet/IP umożliwia 2 tryby działania: tunelowanie (KNXnet/IP Tunelling) i routing (KNXnet/IP Routing). Tunelowanie jest wykorzystywane do przesyłania wiadomości do magistrali z lokalnej sieci Ethernet lub sieci Internet w celu przesyłania telegramów (jak w naszym zadaniu). Tunelowanie wykorzystuje na przykład oprogramowanie ETS, stworzone przez *KNX Association* do konfigurowania sieci KNX i programowania urządzeń w tej sieci. Routing służy do przesyłania telegramów między dwoma strefami KNX połączonymi siecią Ethernet i jest poza zakresem skryptu.

Tunelowanie wymaga nazwania połączenia między źródłem pakietów, a urządzeniem docelowym. W naszym przypadku urządzeniem źródłowym jest nasz komputer, zaś docelowym router KNXnet/IP. Podczas tunelowania wykorzystuje się dwie usługi (pole Service Type Identifier wspomniane wcześniej): 0x0420 (TUNNELLING_REQUEST), który jest żądaniem tunelowania danych

oraz 0x0421 (TUNNELLING_ACK), który jest potwierdzeniem otrzymania żądania.

Wracając do struktury telegramu, pole KNXnet/IP Body składa się z 2 elementów: Connection Header (2 bajty) i cEMI frame. Connection Header zawiera Message Code (1 bajt) oraz Additional Length Information (1 bajt).

Powyższy opis pola KNX-IP/Body od razu nasuwa pytanie, czym są pola Message Code, Additional Length Information oraz cEMI Frame. Niestety w oficjalnej dokumentacji nie mogę tego znaleźć, więc będziemy musieli skorzystać z innego źródła.

Nie zawsze uda się znaleźć informacje, której potrzebujemy w oficjalnej dokumentacji, ponieważ na przykład nie mamy dostępu do pełnej dokumentacji albo informacja jest pominięta. Wtedy pozostają do dyspozycji inne źródła, do których możemy zaliczyć: istniejące programy lub skrypty, które implementują protokół, nieoficjalne źródła znalezione w Internecie, czy ruch przechwycony w sieci i jego analiza (Wireshark posiada wbudowaną analizę wielu protokołów, w tym również KNXnet/IP).

W dokumencie *KNX IP Communication* opis pola KNXnet/IP Body (w przypadku tunelowania) sprowadza się do danych, które opisałem powyżej oraz do diagramu z uzupełnionym przykładem wiadomości. Ponadto jest dodana informacja, że ramka cEMI właściwie zawiera strukturę TP1. Z dokumentu *KNX System Specifications - Architecture* (dostępny w repozytorium) dowiadujemy się, że TP1 to medium wykorzystywane w trybie KNX Twisted Pair, opisany wcześniej.

W dokumencie *KNX IP Communication* brakuje jednak wyjaśnienia poszczególnych pól struktury TP1 vel ramki cEMI. W związku z tym musimy skorzystać z niestandardowego źródła.

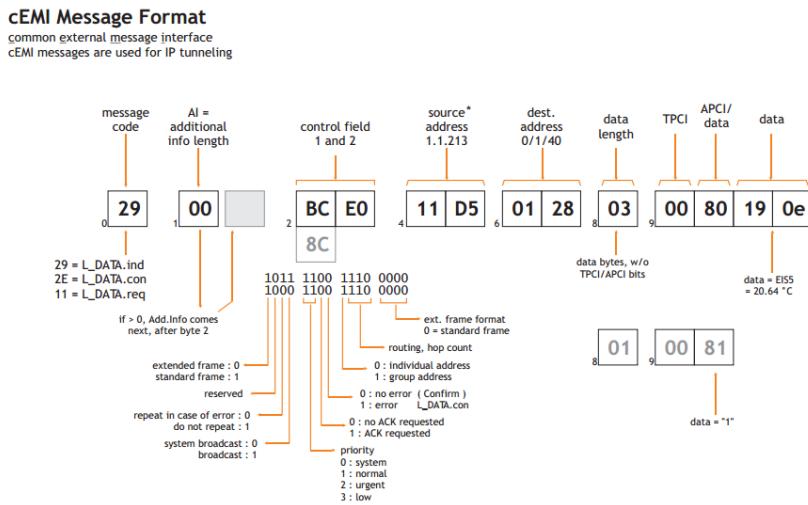
Ramka cEMI

Przykład przedstawiony w *KNX IP Communication* staje się bardziej jasny, gdy skorzystamy z dokumentu ze strony <http://www.dehof.de/eib/pdfs/EMI-FT12-Message-Format.pdf> (dostępny w repozytorium), który umieszczono na Rysunku 4.17. Widać na nim, że pole Message Code może przyjąć następujące wartości:

- 11_{hex} (L_DATA.req), zwykle oznaczające prośbę o zmianę wartości,
- 29_{hex} (L_DATA.ind), zwykle oznaczające poinformowanie o zmianie,
- $2E_{hex}$ (L_DATA.con), zwykle oznaczające potwierdzenie otrzymania ramki.

Oprócz powyższych wartości występują również inne, które są wykorzystywane do transmisji danych pomiędzy warstwą dostępu do sieci, a warstwą sieci, jednakże wykraczają one poza ten prosty przykład.

Kolejne pole to Additional Information Length, które określa długość nagłówka z dodatkowymi informacjami, występującego zaraz za omawianym polem. Zwykle wartość tego pola będzie równa 0.



Rysunek 4.17: KNX - Ramka cEMI

Kolejne 2 bajty to 16 bitów kontrolnych, w których określa się takie wartości jak rodzaj ramki (standardowa, rozszerzona), obsługa błędów (ponów wysłanie w razie błędu), priorytet wiadomości (0-3, gdzie 0 to wiadomość o najwyższym priorytecie), typ adresowania (indywidualny lub grupowy) i inne. Najczęściej, w komunikacji z routerem będą występować bajty *BCC0*, czyli standardowa ramka, bez ponownego wysyłania, o niskim priorytecie z adresowaniem grupowym.

Po bitach kontrolnych w ramce umieszczony jest jej adres źródłowy (2 bajty) oraz docelowy (kolejne 2 bajty). Adresy mogą być indywidualne, o których wspominałem wcześniej (strefa, linia i numer urządzenia) lub grupowe, do których jeszcze wróćmy.

Po adresach umieszczone są dane, czyli ich długość, wartości TPCI (Transport Layer Protocol Control Information), APCI (Application Protocol Control Information) oraz same dane. Oprócz długości danych pole nie są wyrównane do rozmiarów bajtu. I tak pierwszy bajt to rozmiar danych w bajtach, nie wliczający wartości TPCI i APCI. Po nim kolejne 2 bity to bity TPCI (zwykle wartość 0), kolejne 4 są zarezerwowane, następne 4 to APCI, a reszta to same dane.

Rozkład bitów na poszczególne pola ładnie widać na Rysunku 4.18.

Adresowanie

Zakup i połączenie urządzeń wspierających protokół KNX nie wystarczy do ich uruchomienia. Należy jeszcze je zaprogramować, na przykład za pomocą oprogramowania ETS dostarczanego przez KNX Association. Wdrożeniowiec musi wykonać następujące kroki procedury wdrożenia:

```

▼ cEMI
  messagecode: L_Data.ind (0x29)
  add information length: 0 octets
  ▶ Controlfield 1: 0xbc
  ▶ Controlfield 2: 0xc0
  Source Address 0..0..140
  Destination Address 0/0/1 or 0/1
  NPDU length: 1 octets
  00... .... = TPCI: UDT (Unnumbered Data Packet) (0x00)
  .... ..00 10... .... = ACPI: A_GroupValue_Write (0x0002)
  ..00 0000 = Data: 0x00

```

Rysunek 4.18: KNX - Ramka cEMI - Wartość TPCI i ACPI oraz dane

- przydzielenie indywidualnego adresu każdemu z urządzeń,
- przydzielenie grupowego adresu każdemu z urządzeń,
- wybór i instalacja odpowiedniego programu na urządzeniu.

Po zaprogramowaniu urządzeń mogą one się ze sobą komunikować. Na przykład, gdy stan przełącznika zostanie zmieniony na "1", wysyła on telegram, który zawiera jego adres grupowy, wartość "1" oraz dodatkowe dane. Telegram jest odbierany i analizowany przez wszystkie połączone urządzenia, zaś wszystkie urządzenia, które posiadają ten sam adres grupowy odsyłają telegram potwierdzający oraz pobierają aktualną wartość i odpowiednio na nią reagują, np. zamkając obwód.

Indywidualny adres

Adres indywidualny jest unikalny dla każdego urządzenia w sieci i składa się z 2 bajtów podzielonych na 3 części. Pierwsze 4 bity określają numer strefy (ang. area), kolejne 4 bity to numer lini (ang. line), a ostatnie 8 bitów to numer urządzenia. Adres indywidualny jest wykorzystywany w przypadku komunikacji z konkretnym urządzeniem, na przykład podczas diagnostyki, analizy błędów, czy przeprogramowania.

Grupowy adres

Adres grupowy jest wykorzystywany do połączenia szeregu urządzeń w sieci tak, aby reagowały bezpośrednio na zmiany swojego stanu. Na przykład przełącznikowi światła i wszystkim żarówkom (konkretniej pewnie gniazdom na żarówki) przypisujemy ten sam adres grupowy, żeby wszystkie żarówki reagowały na zmianę stanu przełącznika. Podczas działania konfigurowanej sieci urządzenia wykorzystują adresy grupowe, a nie indywidualne.

Adres grupowy może być skonfigurowany jako 2-poziomowy (grupa główna [5 bitów] i podgrupa [11 bitów]), 3-poziomowy (grupa główna [5 bitów], środkowa [3 bitów] i podgrupa [8 bitów]) lub jako wolna struktura. Adres 0/0/0 jest zarezerwowany na adres rozgłoszeniowy do wszystkich urządzeń.

Przykładem konfiguracji adresów grupowych może być następujące rozłożenie poziomów:

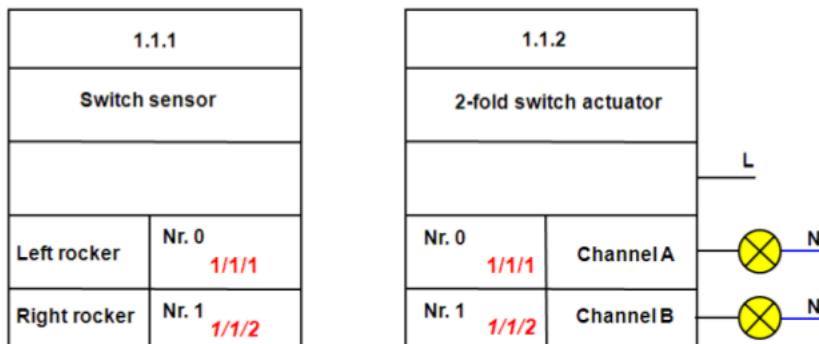
- grupa główna - piętro w budynku,

- grupa środkowa - funkcja (przełączniki, wygaszacz),
- podgrupa - funkcjonalność w konkretnym miejscu (przełączanie światła w kuchni, wygaszanie światła w pokoju).

Każdy aktor, niezależnie od umiejscowienia w sieci, może być połączony do wielu adresów grupowych (np. przełączanie światła w kuchni oraz wygaszanie światła w kuchni) i oczekiwany na wiadomości dla każdego z nich. Natomiast sensory (w tym przełączniki) mogą wysyłać jedynie jeden grupowy adres docelowy w telegramie.

W każdym urządzeniu adresy grupowe są przypisywane do tzw. obiektu grupowego. Powiązanie to jest realizowane za pomocą programu konfigurującego ETS. Obiekt grupowy reprezentuje fragment pamięci w urządzeniach, czyli przechowywane przez urządzenia dane, które mają różny rozmiar. Na przykład informacja o włączeniu lub wyłączeniu przełącznika wymaga 1 bitu, ale informacja tekstowa wymaga wiele bajtów. Maksymalny rozmiar obiektu to 14 bajtów.

Wiele adresów grupowych może być przypisanych do obiektu grupowego w urządzeniu, ale podczas przesyłania zaktualizowanego stanu obiektu grupowego występuje tylko jeden grupowy adres nadawczy. Proces aktualizacji i przesyłania stanu sensora (powiązany z urządzeniami na Rysunku 4.19) jest następujący:



Rysunek 4.19: KNX - Obiekty grupowe

- W lewym przełączniku (ang. left rocker) podwójnego włącznika wcisnięty zostaje przycisk włączenia. Włącznik ustawia wartość "1" w obiekcie grupowym nr 0 i ustawiona zostaje flaga transmisji, oznaczająca potrzebę rozesłania informacji o zmianie stanu obiektu grupowego.
- Obiekt grupowy nr 0 we włączniku jest powiązany z adresem grupowym 1/1/1, dlatego włącznik rozsła telegram w sieci KNX z informacją, że Adres grupowy 1/1/1, ustaw wartość 1".
- Wszystkie urządzenia, które są przypisane do adresu grupowego 1/1/1 wpisują wartość "1" we własnych obiektach grupowych. Na Rysunku 4.19

wartość "1" jest wpisywana w obiekcie numer 0 w aktorze zarządzającym dwiema żarówkami.

- Aplikacja zainstalowana w aktorze odnotowuje zmianę wartości obiektu grupowego i rozpoczyna proces zmiany stanu (np. włącza żarówkę).

Podsumowując powyższy podrozdział i wracając do początkowego zadania wiemy, że musimy skorzystać z tunelowania, aby wysłać do routera KNXnet/IP wiadomość, która powinna być przekazana do urządzenia obsługującego żarówkę. Pojawia się jednak pytanie, jaki adres grupowy posiada żarówka. Odpowiedź można uzyskać na 3 sposoby: zapytać inżyniera, który wdrożył sieć KNX albo sprawdzić w projekcie ETS albo wysyłać kolejno na wszystkie możliwe adresy do chwili, aż żarówka się zapali (metoda siłowa).

4.4.3 Analiza ruchu

Jednym z lepszych źródeł informacji odnośnie wykorzystywanego protokołu oraz formatu przesyłanych pakietów jest analiza ruchu sieciowego, w którym ten protokół jest wykorzystywany. Pojawia się jednak pytanie, skąd ten ruch uzyskać. W przypadku instalacji KNX istnieje możliwość skorzystania z monitora telegramów, będącego częścią programu ETS. Wystarczy połączyć się w programie ETS z instalacją (np. z routerem KNXnet/IP), żeby otrzymywać informację o przesyłanych telegramach. Zrzut ekranu z monitora przedstawiono na Rysunku 4.20.

#	Service	Flags	Prio.	Source Address	Source Name	Destination Address	Rout.	Type	DPT	Info	ACK		
											Find	P	V
1410	from bus	S=0	Low	1.1.100	Device XY	0/7/7	6	Write	05 02 03 04 05 06	LL,ACK			
1411	from bus	S=2	Low	1.1.100	Device XY	0/7/7	6	Write	06 02 03 04 05 06 07 08	LL,ACK			
1412	from bus	S=4	Low	1.1.100	Device XY	0/7/7	6	Write	07 02 03 04 05 06 07 08 09 0A	LL,ACK			
1413	from bus	S=6	Low	1.1.100	Device XY	0/7/7	6	Write	08 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E				
1414	from bus	S=0	Low	1.1.100	Device XY	0/7/7	6	Write	\$00 Off	LL,ACK			
1415	from bus	S=2	Low	1.1.100	Device XY	0/7/4	6	Write	Snow Alarm On	LL,ACK			
1416	from bus	S=4	Low	1.1.100	Device XY	0/7/7	6	Write	02 02 5,14	LL,ACK			
1417	from bus	S=6	Low	1.1.100	Device XY	0/7/7	6	Write	03 02 03	LL,ACK			
1418	from bus	S=0	Low	1.1.100	Device XY	0/7/7	6	Write	04 02 03 04 1.528281E-36	LL,ACK			
1419	from bus	S=2	System	1.1.232	Dimming Control	0/0/0	6	IndividualAdr					
1420	from bus	L=3	Low	1.1.100	Device XY	0/7/7	6	Write	05 02 03 04 05 06	LL,ACK			

Rysunek 4.20: KNX - Monitor magistrali

W naszym przypadku instalacja KNX jest wdrożona, więc istnieje możliwość skorzystania z monitora magistrali. Nie wiemy jednak, czy w sieci KNX są już wysyłane jakieś telegramy, ale to nie jest problem, ponieważ możemy ich wygenerowanie i przesłanie wymusić samemu, na przykład włączając lub wyłączając jeden z przełączników. Spowoduje to wysłanie telegramu na adres grupowy, który jest przypisany do przełącznika. Sam monitor nie przedstawi nam formatu przesyłanego telegramu, ani wartości we wszystkich polach, ale w tym celu możemy skorzystać z Wiresharka. Wystarczy uruchomić nasłuchiwanie na interfejsie, który jest wykorzystywany przez monitor magistrali do połączenia się z routerem KNXnet/IP.

W ramach ćwiczeń można skorzystać ze skryptu symulującego serwer wystawiony przez KNX. Można go uruchomić z folderu `/knx` komendą `python3 knx-server.py 3671 0/0/1`. Nie jesteśmy jednak w stanie podejrzeć żadnych telegramów za pomocą monitora magistrali, ponieważ nie są one przesyłane. Skrypt służy jedynie do sprawdzania reakcji na wysłane przez naszego klienta pakiety.

W repozytorium w pliku `dump.pcapng` znajduje się zrzut ruchu sieciowego, w którym znajduje się również ruch w ramach protokołu KNXnet/IP.

W sytuacji, gdy nie mamy monitora magistrali lub, gdy w sieci KNX nie są przesyłane żadne telegramy i nie możemy ich wymusić możemy skorzystać z istniejących zrzutów ruchu sieciowego lub istniejących programów implementujących interesujący nas protokół, na przykład skanerów sieci KNX. Program `nmap` posiada skrypt do skanowania KNX.

31 79.446330885 81.2.254.217	172.17.0.2	KNXnet..	62 CONNECT_RESPONSE 3671 > 3672
33 85.496978935 172.17.0.1	172.17.0.2	KNXnet..	56 DESCRIPTION_REQUEST 52716 > 3671
34 85.497150103 172.17.0.2	172.17.0.1	KNXnet..	112 DESCRIPTION_RESPONSE 3671 > 52716
35 87.727687579 172.17.0.2	81.2.254.217	KNXnet..	58 DISCONNECT_REQUEST 3672 > 3671
37 87.795987776 81.2.254.217	172.17.0.2	KNXnet..	59 DISCONNECT_RESPONSE 3671 > 3672
38 87.796210646 172.17.0.2	81.2.254.217	ICMP	78 Destination unreachable (Port unreachable)
49 88.889822876 172.17.0.2	81.2.254.217	KNXnet..	68 CONNECT_REQUEST 3672 > 3671
41 89.030549871 81.2.254.217	172.17.0.2	KNXnet..	62 CONNECT_RESPONSE 3671 > 3672
48 112.761486165 172.17.0.1	172.17.0.2	KNXnet..	56 DESCRIPTION_REQUEST 36036 > 3671
49 115.61513972 172.17.0.2	172.17.0.1	ICMP	81 Destination unreachable (Port unreachable)
54 119.053446444 172.17.0.2	81.2.254.217	KNXnet..	58 CONNECTIONSTATE_REQUEST 3672 > 3671
55 119.195709289 81.2.254.217	172.17.0.2	KNXnet..	59 CONNECTIONSTATE_RESPONSE 3674 > 3672
56 125.486489574 172.17.0.1	172.17.0.2	KNXnet..	56 DESCRIPTION_REQUEST 54686 > 3672
57 127.437933755 172.17.0.1	172.17.0.2	KNXnet..	56 DESCRIPTION_REQUEST 47534 > 3672
58 147.597854444 172.17.0.2	81.2.254.217	KNXnet..	63 TUNNELLING_REQUEST 3672 > 3671 cEMI: L_Data.req
59 147.668591341 81.2.254.217	172.17.0.2	KNXnet..	52 TUNNELLING_ACK 3671 > 3672
60 147.668634789 81.2.254.217	172.17.0.2	KNXnet..	63 TUNNELLING_REQUEST 3671 > 3672 cEMI: L_Data.con
61 147.669263735 172.17.0.2	81.2.254.217	KNXnet..	52 TUNNELLING_ACK 3672 > 3671
62 149.095145285 172.17.0.2	81.2.254.217	KNXnet..	56 CONNECTIONSTATE_REQUEST 3672 > 3671
63 149.140691113 81.2.254.217	172.17.0.2	KNXnet..	56 CONNECTIONSTATE_RESPONSE 3671 > 3672
64 161.325766686 172.17.0.2	81.2.254.217	KNXnet..	63 TUNNELLING_REQUEST 3672 > 3671 cEMI: L_Data.req
65 161.463465788 81.2.254.217	172.17.0.2	KNXnet..	52 TUNNELLING_ACK 3671 > 3672
66 161.463514957 81.2.254.217	172.17.0.2	KNXnet..	63 TUNNELLING_REQUEST 3671 > 3672 cEMI: L_Data.con
67 161.464256449 172.17.0.2	81.2.254.217	KNXnet..	52 TUNNELLING_ACK 3672 > 3671
68 162.424812567 81.2.254.217	172.17.0.2	KNXnet..	63 TUNNELLING_REQUEST 3671 > 3672 cEMI: L_Data.ind
69 162.425487629 172.17.0.2	81.2.254.217	KNXnet..	52 TUNNELLING_ACK 3672 > 3671
72 179.071709726 172.17.0.2	81.2.254.217	KNXnet..	56 CONNECTIONSTATE_REQUEST 3672 > 3671
73 179.142826588 81.2.254.217	172.17.0.2	KNXnet..	59 CONNECTIONSTATE_RESPONSE 3671 > 3672
76 193.811363859 172.17.0.1	172.17.0.2	KNXnet..	56 DESCRIPTION_REQUEST 47136 > 3672
77 195.817176127 172.17.0.1	172.17.0.2	KNXnet..	56 DESCRIPTION_REQUEST 56774 > 3672
78 197.822352810 172.17.0.1	172.17.0.2	KNXnet..	56 DESCRIPTION_REQUEST 38045 > 3672

Rysunek 4.21: KNX - Zrzut datagramów w Wiresharku

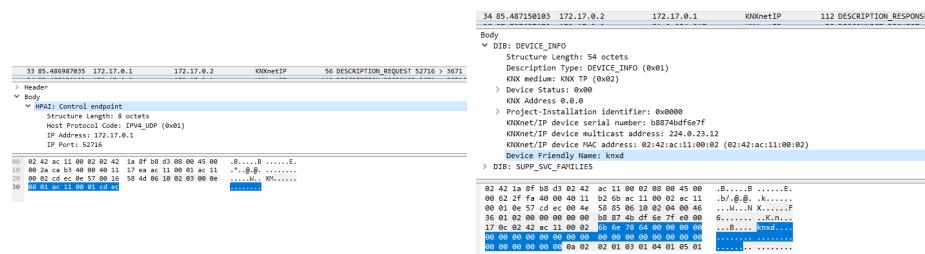
Na Rysunku 4.21 znajduje się zrzut ekranu z komunikacji między klientem KNXnet/IP (dowolne urządzenie), a routerem KNXnet/IP. Widać na nim rozkodowane przez Wireshark datagramy, które w kolumnie `Info` zawierają informację o identyfikatorze akcji (`Service Type Identifier`, patrz rozdział 4.4.2). Można z nich wywnioskować, w jakiej kolejności przesyłane są pakiety oraz które z nich są najpopularniejsze. Ich format zostanie opisany po którym wprowadzeniu.

Na początku (pakiet nr 33) wysyłany jest pakiet `DESCRIPTION_REQUEST` (na domyślny port 3671), w którym klient prosi o przedstawienie się przez router. Odpowiedź otrzymuje w następnym datagramie (`DESCRIPTION_RESPONSE`). Pakiety te są przesyłane w celu zebrania podstawowych informacji o usłudze wystawionej przez router (np. o wersji udostępnionych usług). Ponadto mogą być wykorzystane do sprawdzenia, czy usługa działa na wybranym porcie (skanowa-

nie portów). Jeśli działa, to odpowie pakietem DESCRIPTION_RESPONSE. Kolejne pakiety (nr 40 i 41) to żądanie nawiązania połączenia (CONNECT_REQUEST) i odpowiedź na nie (CONNECT_RESPONSE). Wynika to z faktu, że pozostałe pakiety (w tym tunelowanie) wymagają przesyłania identyfikatora kanału komunikacji, które jest zwracane w pakiecie CONNECT_RESPONSE. Po uzyskaniu identyfikatora klient w każdej chwili może odpytać o stan połączenia (kanału komunikacji) wysyłając żądanie CONNECTIONSTATE_REQUEST. W odpowiedzi otrzymuje CONNECTIONSTATE_RESPONSE, w którym router zwraca informację, czy dany kanał komunikacji funkcjonuje poprawnie.

Dopiero w pakietach korzystających z usługi TUNNELLING_REQUEST przesyłane jest żądanie zmiany wartości przypisanej do wybranego adresu grupowego. W pakiecie nr 58 klient wysyła żądanie zmiany wartości (L_Data.req) przypisanej do adresu 0/0/1 na wartość 0. Kolejnym pakietem jest pakiet TUNNELLING_ACK, odesłany przez router jako potwierdzenie otrzymania pochodzącego pakietu zgodnie z założeniem tunelowania. Następnie w pakiecie nr 60 router odsyła potwierdzenie obsłużenia żądania zmiany wartości (L_Data.con), po czym klient wysyła potwierdzenie otrzymania pakietu (TUNNELLING_ACK). Podobna wymiana pakietów znajduje się w pakietach o numerach 64-69. Tam jednak dochodzi jeszcze jedna para pakietów, czyli pakiet nr 67 przesyłany przez router (L_Data.ind). Oznacza on, że wartość przypisana do adresu grupowego uległa zmianie. W odpowiedzi klient oczywiście odsyła pakiet TUNNELLING_ACK. We wcześniejszej wymianie pakietów nie przesłano L_Data.ind, ponieważ klient zażądał ustalenia wartości adresu grupowego równej wartości obecnej. Stąd, nie nastąpiła zmiana wartości. Na koniec klient przesyła pakiet korzystający z usługi DISCONNECT_REQUEST w celu zamknięcia kanału komunikacyjnego. Router odpowiada potwierdzającym pakietem DISCONNECT_RESPONSE.

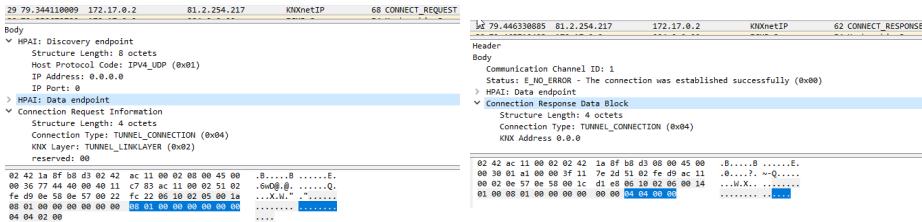
Wszystkie omawiane pakiety korzystają z protokołu UDP, więc omawiany format pominię protokoły warstw niższych niż warstwa aplikacji. Ponadto, wszystkie te pakiety posiadają strukturę zgodną z tą opisaną w rozdziale 4.4.2. Dlatego poniżej omówię jedynie sekcję Body, która jest różna w zależności od wybranego identyfikatora usługi.



Rysunek 4.22: KNX - Pakiety DESCRIPTION_REQUEST oraz DESCRIPTION_RESPONSE

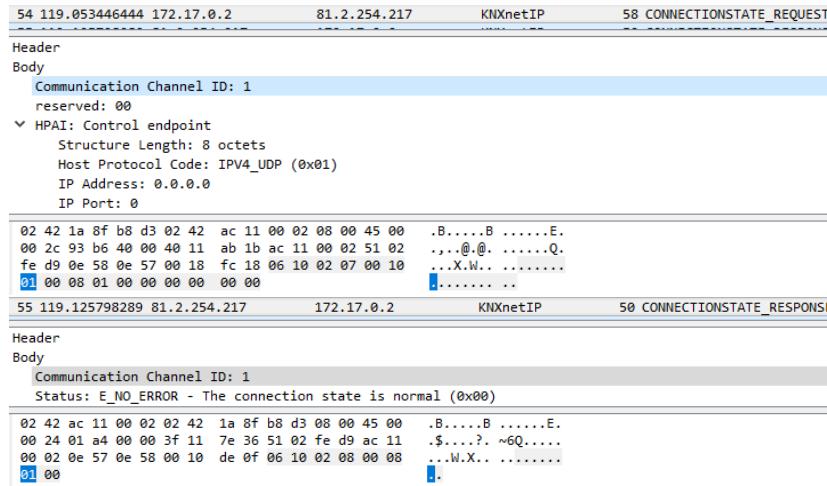
Na Rysunku 4.22 znajdują się pakiety DESCRIPTION_REQUEST oraz DESCRIPTION_RESPONSE. Żądanie w sekcji Body zawiera strukturę HPAI,

która w pierwszym bajcie przechowuje swój rozmiar w bajtach, w drugim określa protokół wykorzystywany przez klienta (wartość 0x01 dla IPV4_UDP), w kolejnych 4 przechowuje adres IP (big endian) i w ostatnich dwóch port (big endian). Adres IP oraz port to dane klienta, na które router powinien odpowiedzieć. Odpowiedź routera pominiemy, ponieważ nie będzie istotna z punktu widzenia naszego zadania. Zachęcam do sprawdzenia formatu tego pakietu w Wiresharku.



Rysunek 4.23: KNX - Pakiety CONNECT_REQUEST oraz CONNECT_RESPONSE

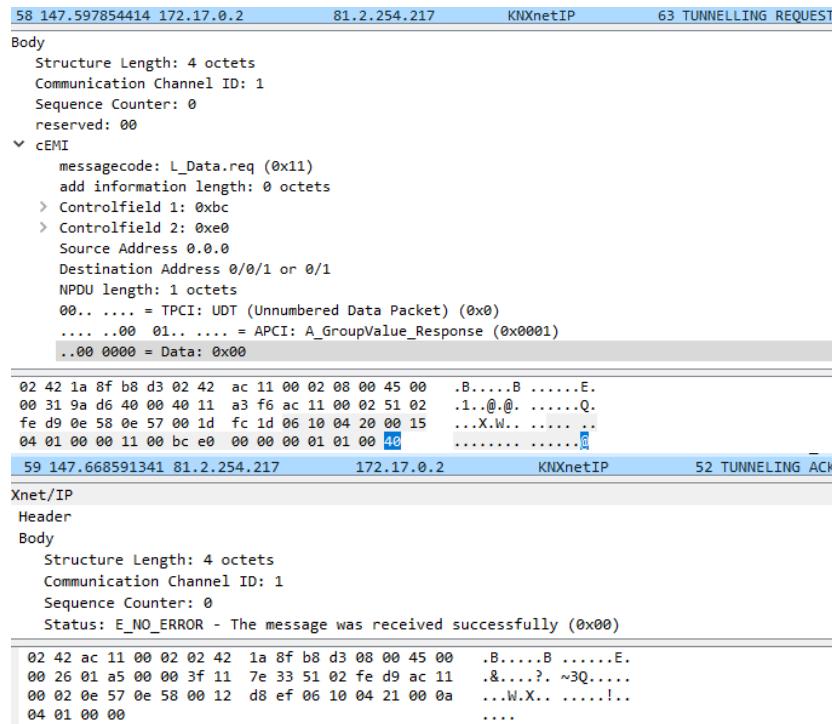
Na Rysunku 4.23 znajdują się pakiety CONNECT_REQUEST oraz CONNECT_RESPONSE. Sekcja Body zapytania składa się z dwóch struktur HPAI, w których adres IP oraz port są równe 0 oraz sekcji Connection Request Information. Sekcja ta zawiera informację o typie połączenia (TUNNEL_CONNECTION) oraz o wykorzystywanej warstwie z protokołu KNX (TUNNEL_LINKLAYER).



Rysunek 4.24: KNX - Pakiety CONNECTIONSTATE_REQUEST oraz CONNECTIONSTATE_RESPONSE

Na Rysunku 4.24 znajdują się pakiety CONNECTIONSTATE_REQUEST oraz CONNECTIONSTATE_RESPONSE. Sekcja Body zapytania składa się z identyfikatora kanału komunikacyjnego (o niego pytamy) oraz struktury HPAI, która

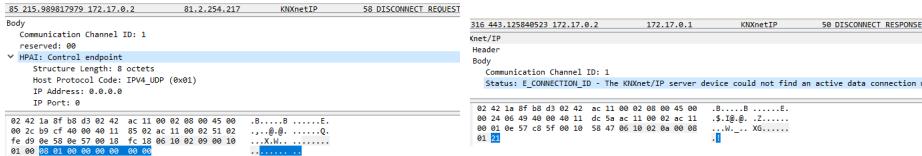
może być „wyzerowana” (zerowy adres IP oraz port). W sekcji Body odpowiadzi są 2 bajty. Pierwszy to identyfikator kanału, zaś drugi to kod stanu. Na rysunku widać, że wartość $0x00_{hex}$ oznacza normalne działanie kanału.



Rysunek 4.25: KNX - Pakiety TUNNELING_REQUEST oraz TUNNELING_ACK

Na Rysunku 4.25 znajdują się pakiety TUNNELING_REQUEST oraz pakiet TUNNELING_ACK. Pierwszy z nich to najbardziej rozbudowany z omawianych pakietów. Sekcja Body składa się z identyfikatora kanału, numeru sekwencyjnego oraz ramki cEMI. Ramka ma format zgodny z tym, który omówiliśmy wcześniej. Message Code ma wartość $0x11_{hex}$, czyli wysyłamy żądanie zmiany wartości. Dalsze kilka bajtów to wartości, które możemy uznać za stałe i przepisać do naszego klienta KNX. Adres źródłowy został wcześniej zwrócony w pakiecie DESCRIPTION_RESPONSE, ale zwykle jest to adres 0.0.0, zakodowany oczywiście na dwóch bajtach. Adres docelowy to 0/0/1 również zakodowany na 2 bajtach. Następny bajt to rozmiar NPDU, czyli sekcji z danymi. Na ostatnich dwóch bajtach znajdują się 3 informacje: pierwsze to informacja o nienumerowanych pakietach (nieistotna z punktu widzenia klienta), druga to typ wiadomości APCI (4 bity na przełomie bajtów), zaś ostatnia, czyli 6 ostatnich bitów to dane. Jeśli rozmiar NDPU byłby większy, niż 1, to kolejne bajty byłyby bajtami danych.

Sekcja Body odpowiedzi zawiera identyfikator kanału, numer sekwencyjny odpowiedzi oraz status odpowiedzi. Wartość $0x00_{hex}$ oznacza, że wiadomość została odebrana pomyślnie.



Rysunek 4.26: KNX - Pakiety DISCONNECT_REQUEST oraz DISCONNECT_RESPONSE

Na Rysunku 4.26 znajdują się pakiety DISCONNECT_REQUEST oraz DISCONNECT_RESPONSE. Sekcja Body zapytania składa się z identyfikatora kanału komunikacyjnego oraz „wyzerowanej” struktury HPAI. W odpowiedzi router odsyła identyfikator kanału komunikacyjnego oraz status obsługi żądania. Na rysunku jest przedstawiona sytuacja, w której router odpowiada statusem $0x21_{hex}$, który oznacza, że router nie mógł odnaleźć połączenia o podanym identyfikatorze.

4.4.4 Budowa klienta KNXnet/IP

Znając typy oraz format pakietów KNX oraz wiedząc, w jakiej kolejności należy je przesyłać (czyli po prostu znając protokół) możemy powrócić do treści zadania do wykonania.

W domu zainstalowano system czujników i detektorów oraz zintegrowany system zarządzania wszystkimi znajdującymi się w budynku instalacjami. Wszystkie urządzenia komunikują się za pomocą protokołu KNX. Ponadto, w domowej sieci istnieje router KNX, który pozwala na zarządzanie instalacjami z domowej sieci IP. Napisz program, który umożliwi zarządzanie urządzeniami w sieci KNX, np. zapalanie żarówek.

Napiszemy protokół, który będzie przyjmował adres IP i port routera, a także adres grupowy KNX oraz wartość 0 lub 1, która reprezentuje włączenie lub wyłączenie aktorów powiązanych z adresem grupowym. Przykład wywołania skryptu to `python knx-client.py 1.2.3.4 3671 0/1/1 1`.

Pierwszym krokiem, który wykona skrypt będzie wysłanie pakietu DESCRIPTION_REQUEST w celu sprawdzenia, czy router KNXnet/IP działa i jest dostępny pod danym portem. W dalszej kolejności będziemy musieli nawiązać połączenie za pomocą pakietu CONNECT_REQUEST, ponieważ pakiety TUNNELLING_REQUEST wymagają identyfikatora kanału komunikacji.

W ramach tunelowania wyślemy ramkę cEMI, w której umieszczone zostaną adres grupowy oraz wartość podane w argumentach. Wartość pola Message Code będzie równa 11_{hex} (L_DATA.req), co oznacza chęć zmiany wartości pod wybranym adresem grupowym.

W odpowiedzi najpierw powinniśmy otrzymać potwierdzenie poprawnego tunelowania TUNNELLING_ACK, po czym powinna się pojawić tunelowana ramka

cEMI z polem Message Code równym $2E_{hex}$ (L_DATA.con), co oznacza potwierdzenie otrzymania ramki z żądaniem zmiany. Jeśli wartość przypisana do wybranego adresu grupowego faktycznie uległa zmianie, to po chwili powinna się pojawić jeszcze jedna tunelowana ramka cEMI z polem Message Code równym 29_{hex} (L_DATA.ind). Oczywiście weryfikacja, czy nasz skrypt działa poprawnie mogłaby polegać na sprawdzeniu, czy żarówka się zaświeciła.

W kontenerze docker istnieje skrypt knx/knx-server.py, który można uruchomić komendą `python3 knx-server.py 3671 0/0/1`. W tym przypadku nie możemy osobiście zweryfikować, czy żarówka się zaświeciła, dlatego skrypt serwera dodatkowo nasłuchuje na UDP/4444 i w odpowiedzi na dowolną wiadomość odeśle w formie tekstowej aktualną wartość przypisaną do adresu grupowego. Na przykład komenda `nc -u 127.0.0.1 4444 i •Enter` pozwala sprawdzić aktualną wartość.

Poniżej Listingu 4.45 znajduje się klient KNX zgodny z powyższymi założeniami.

```

1 import socket, struct, sys, math
2
3
4 class KnxMessage:
5
6     def __init__(self, message=None, **kwargs):
7
8         if message is not None:
9             self._unpack_from_message(message)
10        else:
11            self._unpack_from_kwargs(**kwargs)
12
13     def _unpack_from_message(self, message):
14         service_type = struct.unpack('>H', message[2:4])[0]
15         if service_type != self.get_service_type():
16             print('Protocol error')
17             exit(2)
18
19         self._unpack_body_from_message(message)
20
21     def _unpack_from_kwargs(self, **kwargs):
22         raise NotImplemented()
23
24     def get_service_type(self):
25         raise NotImplemented()
26
27     def build_header(self, body_size):
28         total_size = body_size + 6
29         total_size_bytes = bytes([0x00, total_size]) if total_size < 252 else bytes([0
30             xff, total_size - 252])
31         return bytes.fromhex('0610') + struct.pack('>H', self.get_service_type()) +
32             total_size_bytes
33
34     def build_body(self):
35         raise NotImplemented()
36
37     def get_bytes(self):
38         body = self.build_body()
39         return self.build_header(len(body)) + body

```

```

40 class DescriptionRequestMessage(KnxMessage):
41
42     def __init__(self, message=None, sock_addr=None):
43         super(DescriptionRequestMessage, self).__init__(message=message, sock_addr=
44             sock_addr)
45
46     def _unpack_from_kwargs(self, sock_addr=None):
47         self.sock_addr = sock_addr
48
49     def get_service_type(self):
50         return 0x0203
51
52     def build_body(self):
53         return bytes([0x08, 0x01]) + bytes(list(map(int, self.sock_addr[0].split('.')))
54             ) + struct.pack('>H', self.sock_addr[1])
55
56 class DescriptionResponseMessage(KnxMessage):
57
58     def __init__(self, message=None):
59         super(DescriptionResponseMessage, self).__init__(message=message)
60
61     def _unpack_body_from_message(self, message):
62         self.name = ''
63         i = 30
64         while message[i] != 0:
65             self.name += chr(message[i])
66             i += 1
67
68     def get_service_type(self):
69         return 0x0204
70
71 class ConnectRequestMessage(KnxMessage):
72
73     def __init__(self, message=None):
74         super(ConnectRequestMessage, self).__init__(message=message)
75
76     def _unpack_from_kwargs(self):
77         pass
78
79     def get_service_type(self):
80         return 0x0205
81
82     def build_body(self):
83         hpai = bytes([0x08, 0x01, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00])
84         return hpai + hpai + bytes([0x04, 0x04, 0x02, 0x00])
85
86 class ConnectResponseMessage(KnxMessage):
87
88     def __init__(self, message=None):
89         super(ConnectResponseMessage, self).__init__(message=message)
90
91     def _unpack_body_from_message(self, message):
92         self.communication_channel_id = message[6]
93
94     def get_service_type(self):
95         return 0x0206
96
97
98 class TunnellingRequestMessage(KnxMessage):
99
100    def __init__(self, message=None, communication_channel_id=None, sequence_counter
101        =0, cemi=None):
102        super(TunnellingRequestMessage, self).__init__(message=message,
103            communication_channel_id=communication_channel_id,
104            sequence_counter=sequence_counter, cemi=cemi)

```

```

105
106     def _unpack_from_kwargs(self, cemi=None, communication_channel_id=None,
107                             sequence_counter=0):
108         self.communication_channel_id = communication_channel_id
109         self.sequence_counter = sequence_counter
110         self.cemi = cemi
111
112     def _unpack_body_from_message(self, message):
113         self.communication_channel_id = message[7]
114         self.sequence_counter = message[8]
115         self.cemi = {
116             'code': message[10],
117             'source': message[14:15],
118             'destination': message[16:17],
119             'apci': message[19] & 0x03 << 2
120         }
121
122         size = message[18]
123         data = message[20] & 0x3f
124         if size > 1:
125             i = 0
126             while i < size:
127                 data = data << 8
128                 data += message[21 + i]
129                 i += 1
130         self.cemi['data'] = data
131
132     def get_service_type(self):
133         return 0x0420
134
135     def build_body(self):
136         apci_size = math.ceil((len(bin(self.cemi['data']))[2:]) - 6) / 8.0 + 1
137         apci_data = (self.cemi['apci'] << 6) + (self.cemi['data'] >> 8 * (apci_size-1))
138         apci_data_additional = self.cemi['data'] - (self.cemi['data'] >> 8 * (apci_size-1)) << 8 * (apci_size-1)
139         apci_data_additional_bytes = apci_data_additional.to_bytes(apci_size-1, 'big')
140         return bytes([0x04, self.communication_channel_id, self.sequence_counter, 0x00,
141                     self.cemi['code'], 0x00, 0xbc, 0xe0]) \
142                     + self.cemi['source'] + self.cemi['destination'] + bytes([apci_size]) + \
143                     struct.pack('>H', apci_data) \
144                     + apci_data_additional_bytes
145
146     class TunnellingAckMessage(KnxMessage):
147
148         def __init__(self, message=None, communication_channel_id=None, sequence_counter=0):
149             super(TunnellingAckMessage, self).__init__(message=message,
150                                                 communication_channel_id=communication_channel_id, sequence_counter=sequence_counter)
151
152         def _unpack_from_kwargs(self, communication_channel_id=None, sequence_counter=0):
153             :
154             self.communication_channel_id=communication_channel_id
155             self.sequence_counter=sequence_counter
156
157         def _unpack_body_from_message(self, message):
158             pass
159
160         def build_body(self):
161             return bytes([0x04, self.communication_channel_id, self.sequence_counter, 0x00
162                         ])
163
164         def get_service_type(self):
165             return 0x0421
166
167

```

```

164 def parse_group_address(addr):
165     try:
166         parts = list(map(int, addr.split('/')))
167         if len(parts) != 3:
168             return None
169         return '/'.join(map(str, parts))
170     except:
171         return None
172
173 def group_address_to_bytes(addr):
174     parts = list(map(int, addr.split('/')))
175     return bytes([(parts[0] & 0x1f) << 3] + [parts[1] & 0x7], parts[2] & 0xff)
176
177 def individual_address_to_bytes(addr):
178     parts = list(map(int, addr.split('.')))
179     return bytes([(parts[0] & 0x0f) << 4] + [parts[1] & 0x0f], parts[2] & 0xff)
180
181 def parse_ip(ip):
182     try:
183         parts = list(map(int, ip.split('.')))
184         if len(parts) != 4:
185             return None
186         for i in parts:
187             if i < 0 or i > 255:
188                 return None
189         return '.'.join(map(str, parts))
190     except:
191         return None
192
193 def parse_port(port):
194     try:
195         port = int(port)
196         if port >= 2**16 or port < 0:
197             return None
198         return port
199     except:
200         return None
201
202 def parse_value(value):
203     try:
204         value = int(value)
205         if value not in [0, 1, 11, 257]:
206             return None
207         return value
208     except:
209         return None
210
211 def usage():
212     print("usage: %s <ip> <port> <address> <value>" % sys.argv[0])
213
214 if __name__ == '__main__':
215
216     if len(sys.argv) != 5:
217         usage()
218         exit(1)
219
220     ip = parse_ip(sys.argv[1])
221     port = parse_port(sys.argv[2])
222     address = parse_group_address(sys.argv[3])
223     value = parse_value(sys.argv[4])
224
225     if None in [ip, port, address, value]:
226         usage()
227         exit(1)
228
229     s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
230     print('Connecting to', ip, port, '...')
231     s.connect((ip, port))

```

```

232|     print('Connected.')
233|
234|     print('Sending Description Request.')
235|     s.send(DescriptionRequestMessage(sock_addr=s.getsockname()).get_bytes())
236|
237|     message = s.recv(1024)
238|     print('Received Description Response.')
239|     print('KNXnet/IP name:', DescriptionResponseMessage(message=message).name)
240|
241|     print('Sending Connection Request.')
242|     s.send(ConnectRequestMessage().get_bytes())
243|
244|     message = s.recv(1024)
245|     print('Received Connection Response.')
246|     communication_channel_id = ConnectResponseMessage(message=message).
247|         communication_channel_id
248|     print('Communication Channel ID:', communication_channel_id)
249|
250|     print('Sending Tunneling Request.')
251|     APCI_WRITE = 2
252|     cemi = {
253|         'code': 0x11,
254|         'source': individual_address_to_bytes('0.0.0'),
255|         'destination': group_address_to_bytes(address),
256|         'apci': APCI_WRITE,
257|         'data': value
258|     }
259|     s.send(TunnellingRequestMessage(communication_channel_id=
260|         communication_channel_id, sequence_counter=0, cemi=cemi).get_bytes())
261|
262|     message = s.recv(1024)
263|     print('Received Tunnelling Ack.')
264|     TunnellingAckMessage(message=message)
265|
266|     message = s.recv(1024)
267|     print('Received Tunnelling Request (confirmation).')
268|     print('Message code:', hex(TunnellingRequestMessage(message=message).cemi['code']))
269|
270|     print('Sending Tunneling Ack.')
271|     s.send(TunnellingAckMessage(communication_channel_id=communication_channel_id).
272|         get_bytes())
273|
274|     message = s.recv(1024)
275|     print('Received Tunnelling Request (indication).')
276|     print('Message code:', hex(TunnellingRequestMessage(message=message).cemi['code']))
277|
278|     print('Sending Tunneling Ack.')
279|     s.send(TunnellingAckMessage(communication_channel_id=communication_channel_id).
280|         get_bytes())
281|
282|     print('Closing connection...')
283|     s.close()
284|     print('Closed.')
285|
286|     print('Checking value...')
287|     MONITOR_PORT = 3333
288|     s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
289|     s.connect((ip, MONITOR_PORT))
290|     s.send(b'')
291|     print('Value:', s.recv(100).decode('utf8'))
292|     s.close()

```

Listing 4.45: Klient KNX

W ramach komunikacji przesyłane będą pakiety KNX, dlatego w linii 4 de-

finiuję klasę nadzczną reprezentującą wiadomość KNX. W konstruktorze pakiet może otrzymać argument `message` lub listę innych parametrów nazwanych (`kwargs`). Parametr `message` będzie zawierał bajty pakietu, dlatego jeśli zostanie ustalony (różny od `None`), to w linii 9 będzie wywołana funkcja `_unpack_from_message` uzupełniająca pola pakietu na podstawie bajtów. W przeciwnym razie funkcja `_unpack_from_kwargs`, wywołana w linii 11, będzie uzupełniała pola z pozostałych parametrów (`kwargs`).

Funkcja `_unpack_from_message` w klasie `KnxMessage` sprawdza jedyne, czy typ usługi (`service type`) pakietu zgadza się z tym z oczekiwany, pobieranym za pomocą metody `get_service_type`. Następnie wywołuje metodę `_unpack_body_from_message`, która definiowana jest w klasach podrzędnych.

W linii 34 zdefiniowana jest funkcja `get_bytes`, która zwraca reprezentację bajtową pakietu, która będzie przesyłana przez gniazdo. Wywołuje ona funkcję `build_body`, definiowaną w klasach podrzędnych, w celu wygenerowania bajtów reprezentujących treść wiadomości oraz funkcję `build_header` zwierającą reprezentację bajtową nagłówka wiadomości. Funkcja `build_header` jest zdefiniowana w klasie `KnxMessage` (linia 27), ponieważ jedyna część nagłówka zależna od treści wiadomości to jej rozmiar, który jest przekazywany w parametrze. Na końcu, funkcja `get_bytes` zwraca nagłówek połączony z treścią wiadomości.

Konkretnie klasy wiadomości poznamy w trakcie omawiania komunikacji, dlatego przejdźmy teraz do głównego programu (linia 214). W liniach 216-227 sprawdzamy poprawność i parsujemy argumenty skryptu, czyli adres IP routera, port, adres grupowy oraz wartość, którą chcemy ustawić pod tym adresem.

W liniach 229-232 tworzymy gniazdo i łączymy się z routerem KNXnet/IP. Pierwszą wiadomość tworzymy i wysyłamy w linii 235. Tworzymy w niej obiekt klasy `DescriptionRequestMessage`, któremu przekazujemy w parametrze `sock_addr` adres i port naszego gniazda, pobieramy bajty za pomocą funkcji `get_bytes` oraz przesyłamy je za pomocą funkcji `send`. Klasa `DescriptionRequestMessage` zdefiniowana jest w linii 40 i reprezentuje pakiet z żądaniem przedstawienia się przez router. Zgodnie z poprzednimi ustaleniami, typ usługi tego pakietu to $0x203_{hex}$. Natomiast w treści pakietu znajduje się struktura HPAI, w której pierwszy bajt to rozmiar (8), zaś ostatnie 6 bajtów to adres IP i port (big-endian), na którym nasłuchuje nasz klient (pobrane z parametru `sock_addr`).

W dalszym kroku pobieramy maksymalnie 1024 bajty w linii 237. Taka liczba powinna wystarczyć, bo pakiet nie będzie miał więcej bajtów. Jednakże, jeśli chcielibyśmy być dokładni, to pobralibyśmy 6 bajtów, czyli nagłówek pakietu i z ostatnich dwóch odczytalibyśmy całkowity rozmiar pakietu. Następnie pobralibyśmy pozostałą liczbę bajtów.

W linii 239 tworzymy obiekt klasy `DescriptionResponseMessage` przekazując mu odczytane bajty w parametrze `message`. W takiej sytuacji pola klasy uzupełniane są danymi z bajtów. W przypadku tej klasy jedyne co robimy to sprawdzamy, czy `service_type` się zgadza oraz wyciągamy nazwę routera. Nazwa rozpoczyna się w bajcie o indeksie 30, a kończy na pierwszym bajcie,

który ma wartość $0x00_{hex}$ (tak, jak w napisach w języku C).

Po otrzymaniu poprawnej odpowiedzi nawiązujemy połączenie, tworzymy obiekt klasy `ConnectRequestMessage` i wysyłamy jego bajty w linii 242. Klasa jest zdefiniowana w linii 71. W treści pakietu znajdują się dwie "puste" struktury HPAI (wyzerowany adres IP i port) oraz stałe ostatnie 4 bajty, których znaczenie jest w tej chwili nieistotne.

Następnie ponownie pobieramy maksymalnie 1024 bajty i z nich tworzymy obiekt klasy `ConnectResponseMessage`, który reprezentuje odpowiedź na żądanie połączenia (linia 87). W ramach dekodowania wiadomości ponownie sprawdzamy poprawność pola `service_type` oraz wyciągamy identyfikator kanału komunikacyjnego z bajtu o indeksie 6 (linia 93).

W dalszej kolejności będziemy wysyłać ramkę cEMI korzystając z tunelowania. W liniach 251-257 tworzymy słownik reprezentujący ramkę cEMI, w którym umieszczymy kod $0x11_{hex}$ (`L_Data.req`), zakodowany do 2 bajtów i wyzerowany adres źródłowy (reprezentujący router), zakodowany do 2 bajtów adres grupowy z parametrem, typ APCI (oznaczający ustalenie wartości grupowej) oraz wartość do ustalenia. Słownik wraz z identyfikatorem kanału (wcześniej pobranym) oraz numerem sekwencyjnym są przekazywane do stworzenia obiektu klasy `TunnellingRequestMessage` w linii 258. W linii 99 zdefiniowana jest klasa `TunnellingRequestMessage`, zaś w 134 generowana jest treść wiadomości na podstawie przekazanych parametrów, zgodnie z formatem ramki cEMI.

W kolejnym kroku znowu pobieramy maksymalnie 1024 bajty i generujemy z nich obiekt klasy `TunnellingAckMessage`. Z wiadomości nic nie wyciągamy (nie będzie to nam potrzebne), a jedynie sprawdzamy poprawność jej `service_type`.

W tej chwili do routera została już wysłana wiadomość, której zadaniem jest zmiana wartości przypisanej do adresu grupowego. Naszym zadaniem jest odebranie wiadomości potwierdzającej otrzymanie żądania zmiany wartości. W linii 264 pobieramy maksymalnie 1024 bajty, żeby w linii 266 wyciągnąć z nich dane i stworzyć obiekt klasy `TunnellingRequestMessage` i sprawdzić kod wiadomości (`L_Data.con`). W linii 111 jest funkcja `_unpack_body_from_message` klasy `TunnellingRequestMessage`, w której generowany jest słownik reprezentujący ramkę cEMI na podstawie odpowiednich bajtów z parametru `message`. W odpowiedzi wysyłamy potwierdzenie, czyli obiekt klasy `TunnellingAckMessage`, generowany i wysyłany w linii 269.

Powyższą operację (oczekiwanie na `TunnellingRequestMessage` i odesłanie `TunnellingAckMessage`) powtarzamy w liniach 271-276, ponieważ oczekujemy jeszcze na wiadomość informującą o zmianie wartości przypisanej do adresu grupowego (`L_Data.ind`). Właściwie, te dwie operacje nie są wymagane, ponieważ wartość, którą chcemy ustalić może już być przypisana do adresu grupowego. W takiej sytuacji wiadomość `L_Data.ind` nie zostanie wysłana.

Na końcu, w celu sprawdzenia, czy udało się ustawić wartość, łączymy się z serwerem ponownie, ale na porcie 3333 (linie 282-288). Po wysłaniu pustej wiadomości, odbieramy w formie tekstowej aktualną wartość przypisaną do adresu grupowego.

Na Rysunku 4.27 przedstawiono zrzut ekranu z działania powyższego klienta

KNX.

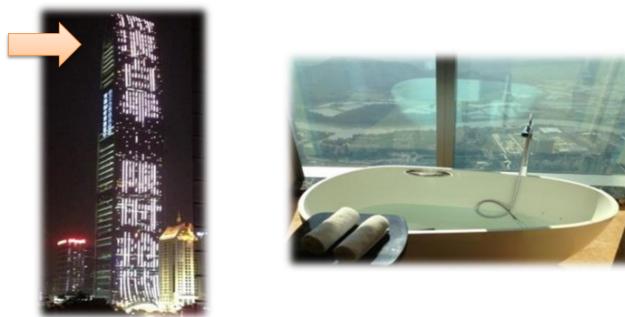
```
root@53aa0e2327f9:/opt/pas/knx# python3 knx-client.py 127.0.0.1 3671 0/1/1 1
Connecting to 127.0.0.1 3671 ...
Connected.
Sending Description Request.
Received Description Response.
KNXnet/IP name: knxd pas
Sending Connection Request.
Received Connection Response.
Communication Channel ID: 1
Sending Tunnelling Request.
Received Tunnelling Ack.
Received Tunnelling Request (confirmation).
Message code: 0x2e
Sending Tunnelling Ack.
Received Tunnelling Request (indication).
Message code: 0x29
Sending Tunneling Ack.
Closing connection...
Closed.
Checking value...
Value: 1
root@53aa0e2327f9:/opt/pas/knx# █
```

Rysunek 4.27: KNX - Zrzut ekranu z klienta KNX

4.4.5 Krótka wzmianka o bezpieczeństwie

Poniższa historia to historia Jesusa Moliny, o której mówił na DefCon 22. Link do filmu na YT znajduje się tutaj: <https://www.youtube.com/watch?v=RX-04XuCW1Y>, a do slajdów tutaj: <https://www.blackhat.com/docs/us-14/materials/us-14-Molina-Learn-How-To-Control-Every-Room-At-A-Luxury-Hotel-Remote.pdf>.

Jesus podróżował po Chinach i zatrzymał się w 5-gwiazdkowym hotelu „The St Regis Shenzhen”.



W pokoju znajdował się iPad do zarządzania urządzeniami obecnymi w pokoju (TV, AC, itp.). Jesus zadał sobie pytanie, czy skoro może kontrolować urządzenia za pomocą iPada, to może to również robić z własnego laptopa. Jak się okazało, iPad jest podłączony do sieci Wifi udostępnionej dla gości, do której

on również miał dostęp. Odpalił więc Wiresharka, żeby pod słuchać komunikację między iPadem i urządzeniami.



Po protokole (3671), który był wykorzystywany do komunikacji z urządzeniami Jesus znalazł informację, że jest to protokół KNX. W tym celu przeprowadził analizę zbliżoną do tej, którą przeprowadziliśmy w tym rozdziale oraz przeanalizował kod open-sourceowych rozwiązań eibd (wcześniej nazwa protokołu KNX), ponieważ komercyjne rozwiązania okazały się bardzo drogie (ok 1000E).

Co więcej, okazało się, że protokół KNX nie jest w ogóle zabezpieczony (analogiczna sytuacja do omawianej w rozdziale). Są co prawda rozszerzenia wprowadzające zabezpieczenia, jak EIBsec (2006 r.) oraz nowy KNX (2013 r.) jednakże ze względu na duży koszt dostępu do specyfikacji, Jesus je pominął. W końcu hotel i tak korzystał z otwartej wersji (bez zabezpieczeń).

Po analizie podsłuchanych pakietów okazało się, że każdy pokój w hotelu ma własny adres IP routera, zaś wszystkie urządzenia są przypisane do różnych adresów grupowych. W dalszej części Jesus opowiada o formacie przesyłanych wiadomości, czyli o czymś, co już znasz.

Ostatecznie Jesus był najprawdopodobniej (nie zweryfikował tego ze względów oczywistych) w stanie zarządzać wszystkimi urządzeniami podłączonymi do magistrali KNX we wszystkich pokojach hotelowych.

Analogiczny test i w konsekwencji włamanie przeprowadzono w budynku „Green Building” na kampusie MIT, w efekcie czego elewacja budynku przerodziła się w grę Tetris (http://hacks.mit.edu/Hacks/by_year/2012/tetris/).

Dalsze kroki...

- Router KNXnet/IP domyślnie nasłuchuje na porcie 3671, jednakże można go uruchomić na dowolnym innym porcie. Jak sprawdzić, na którym porcie działa usługa?
 - Pierwszą parą pakietów wymienianych między klientem, a routerem są żądanie opisu (description request) i odpowiedź na nie.
 - W celu zidentyfikowania, na którym porcie działa usługa można wysyłać żądania na kolejne porty i sprawdzać, czy odesłana została odpowiedź.

4.5 Własny protokół

W poprzednich rozdziałach przeanalizowaliśmy kilka istniejących protokołów, dla których stworzyliśmy klientów oraz dla niektórych serwery. W tym rozdziale stworzymy własne protokoły oraz zaimplementujemy je. Będą to wciąż proste implementacje, które bedziemy rozwijać w dalszej części.

Klasyczny przykład protokołu zakłada, że komunikuje się ze sobą klient oraz serwer. Serwer oczekuje na połączenie i gdy klient się z nim połączy wymieniają się wiadomościami i kończą połączenie. Niektóre protokoły (np. HTTP) zakładają jednorazową wymianę zapytania i odpowiedzi, zaś inne (np. IMAP) wymagają wielokrotniej wymiany wiadomości, ponieważ w jednej nie można zatrzymać pełnego zapytania.

Przed rozpoczęciem projektowania protokołu należy ustalić w jaki sposób klient oraz serwer będą wiedziały, że otrzymały pełną wiadomość od drugiej strony. Należy pamiętać, że połączenie TCP jest połączeniem strumieniowym, czyli niekończącym się strumieniem bajtów, więc trzeba określić sposób oddzielania jednej wiadomości od drugiej.

Problem ten to z angielskiego *framing* i istnieje kilka sposobów radzenia sobie z nim:

- Pierwszy sposób polega na tym, że serwer po otrzymaniu zapytania odsyła odpowiedź i zamyka połączenie.
- Druga opcja to ustalenie stałej długości wiadomości. Wtedy odbiorca odczytuje stałą liczbę bajtów.
- Można również tak skonstruować wiadomości, że na początku każdej z nich umieszczona jest jej długość. Wtedy odbiorca pobiera stałą liczbę bajtów, żeby sprawdzić jak długa jest cała wiadomość, po czym pobiera pozostałą część.
- Ostatnia opcja to ustalenie specjalnego znaku lub kilku znaków, które oznaczają koniec wiadomości. Odbiorca wczytuje wiadomość bajt za bajtem i sprawdza, czy pobrał już separator wiadomości.

Opcja nr 1 to dobre rozwiązanie w przypadku prostego protokołu, jest łatwa w implementacji i nie wymaga specjalnej obsługi separatorów wiadomości.

Jednakże w przypadku tego rozwiązania nie można wymienić wiele wiadomości w ramach jednego połączenia, dlatego jeśli protokół zakłada wymianę wielu wiadomości, wydajność tej opcji znacznie się zmniejsza.

Opcja 2 jest również prosta w implementacji, jednakże zakłada, że każda wiadomość ma stałą długość. Oczywiście można założyć, że jeśli wiadomość jest krótsza niż ustalony rozmiar to można ją uzupełnić do ustalonego rozmiaru według określonego algorytmu (ang. padding). To jednakże nie rozwiązuje problemu, który występuje w przypadku wiadomości, które mogą być dłuższe niż ustalony rozmiar.

Opcja nr 3 jest powszechnie wykorzystywana, pomimo większej trudności implementacji. Jest to rozwiązanie, które optymalizuje ruch sieciowy, a poza tym nie wymaga analizowania odbieranych danych (poza rozmiarem na początku wiadomości). Jednakże sytuacją, w której opcja ta nie jest wydajna jest protokół wykorzystujący krótkie wiadomości, np. jedno-, duznakowe. Wtedy w każdej z nich jeden lub więcej bajtów określających rozmiar zamuje dużo miejsca w stosunku do całej długości wiadomości.

Ostatnia, czwarta opcja jest najbardziej wydajnym rozwiązaniem, pod warunkiem, że separator wiadomości jest jednoznakowy. Wybór separatora wymaga przeanalizowania wiadomości, które mogą być przesyłane, ponieważ separator nie może wystąpić w żadnej z nich. W przeciwnym razie należy znaleźć zestaw znaków oznaczający koniec wiadomości (np. w przypadku potoku SMTP są to \r\n.\r\n). Popularnym znakiem jest znak NULL, którego wartość liczbową to 0. Jest on również wykorzystywany do zakończenia napisów w ANSI C. Po określeniu separatora odbiorca wczytuje bajt po bajcie i sprawdza, czy napotkał separator. Jeśli tak to wie, że pobrał już całą wiadomość.

Po określeniu sposobu rozdzielania wiadomości należy określić również ich przepływy, czyli typy oraz kolejność przekazywania wiadomości między klientem, a serwerem. Klasycznym przykładem jest przepływ, w którym klient łączy się z serwerem, a następnie powtarzana jest sekwencja:

- serwer oczekuje na wiadomość,
- klient wysyła wiadomość i oczekuje na odpowiedź,
- serwer odbiera wiadomość, przetwarza ją i wysyła odpowiedź.

Powyższa sekwencja jest powtarzana aż do momentu, kiedy klient do serwera lub serwer do klienta wyśle wiadomość oznaczającą zakończenie połączenia lub jedna ze stron po prostu zerwie połączenie.

4.5.1 Kółko i krzyżyk

Pierwszy protokół, który zaprojektujemy będzie wykorzystywany do gry w kółko i krzyżyk. Zakładamy, że wiadomości będą miały stały rozmiar, równy 10 bajtów, a wiadomości krótsze będą uzupełniane znakiem spacji. Na przykład wiadomość „START” będzie tak naprawdę równa „START ” (z dodatkowymi 5 spacja).

Ponadto ustalamy następujący przepływ wiadomości:

1. Serwer oczekuje na połączenie.
2. Klient łączy się z serwerem.
3. Serwer oczekuje na wiadomość.
4. Klient wysyła komendę START.
5. Serwer odpowiada OK.
6. Klient wysyła komendę CHECK <nr>, gdzie <nr> to numer pola, na którym chce postawić krzyżyk.
7. Serwer odbiera pole, oznacza krzyżykiem i odsyła OK lub END.
8. Serwer wybiera pole na którym chce postawić kółko i wysyła do klienta CHECK <nr>.
9. Klient odbiera pole i oznacza kółkiem i odsyła OK lub END.
10. Kroki 6-8 są powtarzana dopóki gra nie została zakończona, czyli jedna ze stron wygrała lub nie ma już miejsc do postawienia kółka lub krzyżka. Jeśli gra jest skończona, jedna ze stron odsyła komendę END (zamiast OK) i kończy połączenie.

W naszym protokole będą występować następujące wiadomości:

- START - wysyłana przez klienta w celu rozpoczęcia gry.
- OK - zatwierdzająca przyjęcie wiadomości.
- CHECK <nr> - oznaczająca wybranie danego pola.
- END - kończąca grę.
- ERROR - oznaczająca, że wystąpił błąd (np. gdy chcemy postawić znak na zajętym polu).

Po zaprojektowaniu prostego protokołu możemy się wziąć za implementację, zakładając, że gdy wystąpi błąd to skrypt zrywa połączenie i kończy działanie. Poniżej umieściłem kod serwera i klienta.

```

1 import sys
2 import socket
3 from kikutils import *
4
5 MY_SIGN = 'o'
6 THEIR_SIGN = 'x'
7
8 if __name__ == "__main__":
9
10    if len(sys.argv) != 2:
11        sys.stderr.write("usage: %s port\n" % (sys.argv[0], ))
12        exit(1)
13
14    try:
15        port = int(sys.argv[1])

```

```

16     assert port > 0
17 except:
18     sys.stderr.write("error: invalid port\n")
19     exit(1)
20
21 sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
22 sock.bind(("0.0.0.0", port))
23 sock.listen(5)
24
25 while True:
26     client, addr = sock.accept()
27     board = '?' * 9
28
29     cmd = client.recv(MSG_SIZE).strip()
30     if cmd != 'START':
31         print 'Invalid command'
32         client.send(pad_msg('ERROR'))
33         client.close()
34         continue
35
36     client.send(pad_msg('OK'))
37
38     while True:
39         cmd = client.recv(MSG_SIZE).strip()
40
41         if cmd == 'END' or cmd == '':
42             print_board(board)
43             winner = check_winner(board)
44             if winner is None:
45                 print 'Game ended unexpectedly.'
46             if winner == MY_SIGN:
47                 print 'You won!'
48             else:
49                 print 'You lost!'
50             client.close()
51             break
52
53         if cmd.startswith('CHECK '):
54             try:
55                 fieldnb = int(cmd.split(' ')[1])
56                 if fieldnb < 0 or fieldnb > 8:
57                     raise ValueError
58             except ValueError:
59                 print 'Invalid command'
60                 client.send(pad_msg('ERROR'))
61                 client.close()
62                 break
63
64             if board[fieldnb] != '?':
65                 print 'Field taken'
66                 client.send(pad_msg('ERROR'))
67                 client.close()
68                 break
69
70             board[fieldnb] = THEIR_SIGN
71             winner = check_winner(board)
72             if winner in [MY_SIGN, THEIR_SIGN]:
73                 print_board(board)
74                 if winner == MY_SIGN:
75                     print 'You won!'
76                 else:
77                     print 'You lost!'
78                 client.send(pad_msg('END'))
79                 client.close()
80                 break
81
82             client.send(pad_msg('OK'))
83

```

```

84     while True:
85         print_board(board, numbers=True)
86         try:
87             fieldnb = int(raw_input('What is your field number? '))
88             if fieldnb < 0 or fieldnb > 8:
89                 raise ValueError
90             if board[fieldnb] != '?':
91                 print 'This field is not empty'
92             else:
93                 break
94         except ValueError:
95             print 'Invalid field number'
96             continue
97
98         board[fieldnb] = MY_SIGN
99         client.send(pad_msg('CHECK ' + str(fieldnb)))
100
101        if client.recv(MSG_SIZE).strip() != 'OK':
102            print 'Invalid response:', resp
103            client.close()
104            break
105
106        print 'Invalid command'
107        client.send(pad_msg('ERROR'))
108        client.close()
109        break

```

Listing 4.46: Serwer Kółko i Krzyżyk

```

1 import sys
2 import socket
3 from kikutils import *
4
5 MY_SIGN = 'x'
6 THEIR_SIGN = 'o'
7
8 if __name__ == "__main__":
9
10    if len(sys.argv) != 3:
11        sys.stderr.write("usage: %s ip port\n" % (sys.argv[0], ))
12        exit(1)
13
14    try:
15        addr = sys.argv[1]
16        port = int(sys.argv[2])
17        assert port > 0
18    except:
19        sys.stderr.write("error: invalid port\n")
20        exit(1)
21
22    sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
23    sock.connect((addr, port))
24
25    board = '?' * 9
26    sock.send(pad_msg('START'))
27    resp = sock.recv(MSG_SIZE).strip()
28    if resp != 'OK':
29        print 'Invalid response:', resp
30        sock.close()
31        exit(1)
32
33    while True:
34        # Select field
35        while True:
36            print_board(board, numbers=True)
37            try:
38                fieldnb = int(raw_input('What is your field number? '))

```

```

39     if fieldnb < 0 or fieldnb > 8:
40         raise ValueError
41     if board[fieldnb] != '?':
42         print 'This field is not empty'
43     else:
44         break
45 except ValueError:
46     print 'Invalid field number'
47
48 board[fieldnb] = MY_SIGN
49 sock.send(pad_msg('CHECK ' + str(fieldnb)))
50 if sock.recv(MSG_SIZE).strip() != 'OK':
51     print 'Invalid response:', resp
52     sock.close()
53     break
54
55 cmd = sock.recv(MSG_SIZE).strip()
56
57 if cmd == 'END' or cmd == '':
58     print_board(board)
59     winner = check_winner(board)
60     if winner is None:
61         print 'Game ended unexpectedly.'
62     if winner == MY_SIGN:
63         print 'You won!'
64     else:
65         print 'You lost!'
66     sock.close()
67     break
68
69 if cmd.startswith('CHECK '):
70     try:
71         fieldnb = int(cmd.split(' ')[1])
72         if fieldnb < 0 or fieldnb > 8:
73             raise ValueError
74         except ValueError:
75             print 'Invalid command'
76             sock.send(pad_msg('ERROR'))
77             sock.close()
78             break
79
80     if board[fieldnb] != '?':
81         print 'Field taken'
82         sock.send(pad_msg('ERROR'))
83         sock.close()
84         break
85
86     board[fieldnb] = THEIR_SIGN
87     winner = check_winner(board)
88     if winner in [MY_SIGN, THEIR_SIGN]:
89         print_board(board)
90         if winner == MY_SIGN:
91             print 'You won!'
92         else:
93             print 'You lost!'
94         client.send(pad_msg('END'))
95         client.close()
96         break
97
98     print 'Invalid command'
99     client.send(pad_msg('ERROR'))
100    client.close()
101    break

```

Listing 4.47: Klient Kółko i Krzyżyk

Dalsze kroki...

- Zmodyfikuj protokół oraz implementację w taki sposób, żeby korzystał z opcji nr 3 do rozdzielania wiadomości. Niech w każdej wiadomości pierwszy bajt przesyła rozmiar całej wiadomości (wiadomości nie będą dłuższe niż 255 znaków, wliczając bajt z rozmiarem).
- Zmodyfikuj protokół oraz implementację w taki sposób, żeby odbiorca po sprawdzeniu, że dane pole jest zajęte zwracał informację, że pole jest zajęte i ponownie czekał na nowy numer pola, a nadawca ponownie o ten numer prosił użytkownika. Możesz to zrobić dodając kody do komendy ERROR.
- Korzystając z tego protokołu i implementacji zaprojektuj i zaimplementuj protokół do gry w statki. Możesz przyjąć, że plansza jest wczytywana z pliku zarówno po stronie serwera i klienta (nie trzeba ich przesyłać do serwera).

5

Mechanizmy zaawansowane

W poprzednich rozdziałach wykorzystywaliśmy oraz tworzyliśmy proste skrypty serwera oraz klienta. Zakładały one jawną komunikację, prosty model wymiany wiadomości pt. raz ty, raz ja, a także sekwencyjną obsługę klientów, czyli możliwość obsługi tylko jednego klienta w danej chwili.

W tym rozdziale poznamy zaawansowane mechanizmy, dzięki którym będziemy mogli zabezpieczyć komunikację oraz obsłużyć wiele klientów jednocześnie. Mechanizmy te poznamy na przykładzie skryptu chatu.

5.1 Zaawansowany chat

Na początku rozdziału wprowadzającego do zaawansowanego chatu przypomnijmy sobie prosty, synchroniczny, dwuosobowy chat.

```
1 import sys
2 import socket
3
4 BUF_SIZE = 1024
5
6 if __name__ == "__main__":
7
8     if len(sys.argv) != 2:
9         sys.stderr.write("usage: chat_server port\n")
10        exit(1)
11
12    try:
13        port = int(sys.argv[1])
14        assert port > 0
15    except:
16        sys.stderr.write("error: invalid port\n")
17        exit(1)
18
19    sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
20    sock.bind(("0.0.0.0", port))
21    sock.listen(5)
22
23    while True:
24        client, addr = sock.accept()
25        print(addr[0], "connected now.")
26        print('Receive and send messages. Empty line closes connection.')
27
```

```

28     while True:
29         data = client.recv(BUF_SIZE)
30         if not data:
31             break
32         print 'Msg:', data
33
34         data = raw_input('Answer:')
35         if data == '':
36             break
37         client.send(data)
38
39     client.close()
40
41     sock.close()

```

Listing 5.1: Prosty serwer Chat

Serwer oczekuje na połaczenie i gdy klient się z nim połączy oczekuje na pierwszą wiadomość od klienta i dopiero wtedy będzie mógł odesłać swoją wiadomość, stąd chat jest synchroniczny. Ponadto, serwer jest w stanie obsłużyć tylko jednego klienta, stąd dwuosobowa komunikacja.

```

1 import sys
2 import socket
3
4 BUF_SIZE = 1024
5
6 if __name__ == "__main__":
7
8     if len(sys.argv) != 3:
9         sys.stderr.write("usage: chat_client ip port\n")
10        exit(1)
11
12    try:
13        addr = sys.argv[1]
14        port = int(sys.argv[2])
15        assert port > 0
16    except:
17        sys.stderr.write("error: invalid port\n")
18        exit(1)
19
20    sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
21    try:
22        sock.connect((addr, port))
23
24        print 'Connected...'
25        print 'Send and receive messages. Empty line closes connection.'
26
27        while True:
28            data = raw_input('Msg:')
29            if data == '':
30                break
31            sock.send(data)
32            data = sock.recv(BUF_SIZE)
33            if not data:
34                break
35            print 'Answer:', data
36        sock.close()
37
38    except socket.error, e:
39        print 'Error:', e

```

Listing 5.2: Prosty client Chat

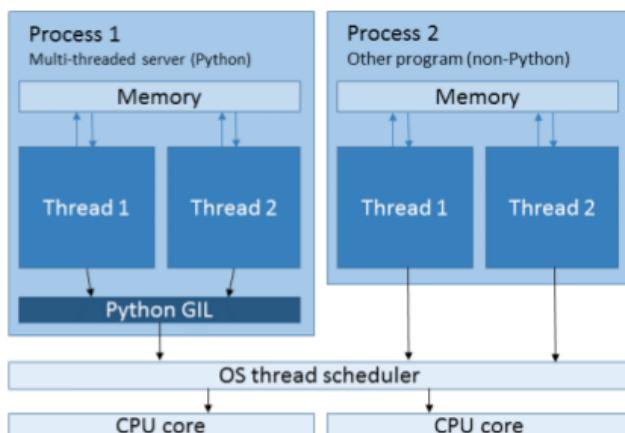
Klient łączy się z serwerem i w pętli najpierw wysyła swoją wiadomość, a później odbiera odpowiedź z serwera.

5.1.1 Wielowątkowość

Podstawowy mechanizm, jaki poznamy na przykładzie zaawansowanego czatu to wielowątkowość. Dzięki niej będziemy mogli rozmawiać z wieloma osobami jednocześnie i nie będziemy musieli czekać na odpowiedź drugiej osoby, żeby wysłać drugą wiadomość. Krótko mówiąc, wielowątkowość pozwala na uruchomienie wielu funkcji równolegle.

Aplikacje można pisać wykorzystując wielowątkowość (ang. multithreading) oraz wieloprocesorowość (ang. multiprocessing). Różnica polega na tym, że funkcje, które chcemy uruchomić równolegle w pierwszym przypadku uruchamiane są w wielu wątkach, a w drugim w wielu procesach.

Wątki można podzielić na „IO bound” i „CPU bound”. Pierwsze z nich to wątki, które oczekują na jakieś dane, np. z gniazda. Drugie to takie, które chcą wykonać instrukcje procesora, więc go potrzebują. Procesor zarządza swoim czasem i przydziela go sprawiedliwie do wszystkich wątków i procesów.

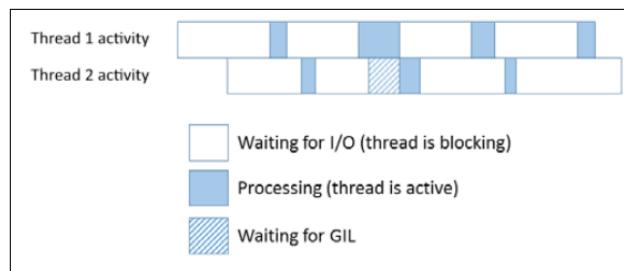


Rysunek 5.1: Wielowątkowość - architektura. Źródło: [7].

Na Rysunku 5.1 znajduje się schemat obsługi wątków. Po lewej stronie jest sytuacja, w której wątki są uruchomione w ramach jednego procesu i wtedy współdzielą pamięć, dzięki czemu mogą się odwoływać do wspólnych zmiennych (shared objects). Po prawej jest sytuacja, w której każdy wątek jest uruchomiony w innym procesie. W takiej sytuacji procesy nie współdzielą pamięci, dla tego w celu komunikacji należy skorzystać z innych mechanizmów, np. gniazd, specjalnie alokowanych obszarów pamięci lub po prostu plików.

Różnica w przypadku języka Python polega na tym, że wykorzystuje on GIL, czyli Global Interpreter Lock. Jego zadanie to upewnienie się, że w danej chwili uruchomiony jest tylko jeden wątek Pythona, nawet jeśli w komputerze jest wiele rdzeni. Plusem wykorzystania GILA jest łatwa obsługa i implementacja interpretera w języku C. Wadą tego rozwiązania jest brak możliwości wykorzystania wielu rdzeni. Nie jest to jednak wada, która znacząco wpływa na aplikację, po-

nieważ większość czasu wątek jest „IO bound”, czyli czeka na wejście/wyjście. Przykład jest przedstawiony na Rysunku 5.2.



Rysunek 5.2: Wielowątkowość - GIL. Źródło: [7].

Do napisania zaawansowanego czatu wykorzystamy wielowątkowość, ponieważ jest ona łatwiejsza w obsłudze (wspólne zmienne dla wątków), a obecność GILA nie będzie nam przeszkadzała, ponieważ większość czasu wątki będą cześć na dane z gniazda.

5.1.2 Protokół chatu

Dodając wielowątkowość do chatu, a tym samym usuwając synchronizację w komunikacji zmienimy sposób działania serwera oraz klienta. Teraz serwer nie będzie aplikacją interaktywną, tylko aplikacją działającą w tle. Jeśli dwie osoby będą chciały ze sobą porozmawiać, to obie będą musiały uruchomić klienta, który połączy się z niezależnym serwerem.

Klient może działać w dwóch trybach. Pierwszy z nich zakłada, że co jakiś czas klient nawiązuje połączenie z serwerem, sprawdza czy pojawiły się nowe wiadomości i je pobiera. Drugi tryb zakłada, że klient nawiązuje i utrzymuje połączenie, dzięki czemu może odebrać wiadomość w tej samej chwili, w której się pojawia. Oba tryby mają swoje wady i zalety i decyzja jest zależna od konkretnej aplikacji. Cykliczne odpytywanie serwera zmniejsza obciążenie serwera, jednakże opóźnia przesłanie wiadomości do klienta. Utrzymywanie połączenia pozwala na zmniejszenie volumenu przesyłanych danych oraz natychmiastowe ich przesyłanie. Wymaga jednak, aby serwer stale obsługiwał połączenie.

W przypadku chatu skorzystamy z trybu, który utrzymuje połączenie. Protokół będzie wyglądał następująco:

1. Protokół będzie wykorzystywał połączenia TCP.
2. Klient będzie łączył się z serwerem, a połączenie będzie utrzymywane.
3. Serwer będzie oczekiwał na połączenia od klientów i odbierał od nich wiadomości.
4. Klient będzie oczekiwał na i odbierał wiadomości od serwera.

5. Serwer będzie wysyłał otrzymaną wiadomość od klienta do wszystkich klientów. Również do nadawcy.
6. Wiadomości będą zakodowane UTF-8 i będą zakończone znakiem NULL (0).
7. Pierwsza wiadomość od klienta do serwera to jego nazwa, którą będzie identyfikowany.
8. Pozostałe wiadomości wysyłane przez klienta to treść wiadomości, która ma być przekazana do wszystkich klientów.
9. Serwer wysyłając wiadomość do klientów będzie dodawał na jej początku nazwę klienta, którą oddzieli od treści znakiem „:”.

Na początek zajmiemy się odbieraniem wiadomości. Nasz protokół zakłada, że znak NULL kończy wiadomość, a wykorzystujemy protokół TCP, czyli strumieniowy. Dlatego musimy napisać funkcje, które pobierają i dekodują wiadomość.

Skrypt będzie napisany w języku Python w wersji 3.

```

1 def parse_recv_data(data):
2     """ Break up raw received data into messages, delimited
3         by null byte """
4     parts = data.split(b'\0')
5     msgs = parts[:-1]
6     rest = parts[-1]
7     return (msgs, rest)
8
9 def recv_msg(sock, data=b''):
10    """ Receive data and break into complete messages on null byte
11        delimiter. Block until at least one message received, then
12        return received messages """
13
14    msgs = []
15    while not msgs:
16        recv = sock.recv(4096)
17        if not recv:
18            raise ConnectionError()
19        data = data + recv
20        (msgs, rest) = parse_recv_data(data)
21    msgs = [msg.decode('utf-8') for msg in msgs]
22    return (msgs, rest)
```

Listing 5.3: Odbieranie wiadomości. Źródło: [7].

Funkcja `recv_msg` przyjmuje dwa parametry. Pierwszy z nich to gniazdo, z którego dane mają być odebrane. Drugi to dane, które zostały pobrane w poprzednim wywołaniu funkcji, ale nie zostały zdekodowane. W połączeniu strumieniowym możemy otrzymać na przykład dane `pierwsza\0druga\0trze`. Są to dwie pełne wiadomości i początek trzeciej. Zadaniem funkcji jest odebranie tych danych, rozkodowanie dwóch pierwszych wiadomości i przekazanie początku trzeciej w drugim wywołaniu funkcji jako drugi parametr. Dzięki temu, początek trzeciej wiadomości nie zostanie pominięty.

W linii 15 znajduje się pętla `while`, która działa, dopóki nie zostaną pobrane i zdekodowane wiadomości w zmiennej `msg`. W pętli funkcja pobiera dane z gniazda (maksymalnie 4096 bajtów). Jeśli nie zostaną obrane żadne dane, to znaczy że połączenie zostało werwane i wyrzucamy wyjątek `ConnectionError`. W przeciwnym razie doposujemy pobrane dane do pozostałych danych z poprzedniego uruchomienia funkcji (`data`) i próbujemy je parsować za pomocą funkcji `parse_recv_data`. Jeśli uda się rozkodować wiadomości to funkcja dekoduje je w UTF-8 (założenie protokołu) oraz zwraca wraz z danymi, których nie udało się rozkodować (*trze* w powyższym przykładzie). Funkcja `parse_recv_data` jedynie rozdziela wiadomości po znaku \0 i wszystkie wiadomości poza ostatnią zapisuje w zmiennej `msg`, zaś ostatnią w zmiennej `rest`. Ostatnia wiadomość to ta, której nie udało się rozkodować, czyli początek nie w pełni odebranej wiadomości. W sytuacji, gdy pobrane zostały pełne wiadomości, na przykład `pierwsza\0druga\0`, zmieniąca `rest` będzie pusta. Na koniec funkcja zwraca wiadomości oraz pozostałość w krotce.

Od tego momentu będziemy wykorzystywali funkcję `recv_msgs` do pobierania danych z gniazda w naszym protokole zamiast metody `recv` przypisanej do gniazda.

5.1.3 Synchronizacja wątków

Drugi problem to obsłużenie wielu klientów równocześnie, czyli wracamy do wielowątkowości. Nasz serwer musi stale nasłuchiwać na połączenia od nowych klientów oraz nasłuchiwać na dane od połączonych już klientów. Dlatego stworzymy funkcję, która obsługuje jednego klienta i dla każdego klienta będziemy ją uruchamiać w oddzielnym wątku. Natomiast w wątku głównym będziemy oczekiwania na nowych klientów.

Na początek omówimy sobie wątek główny, czyli przyjmowanie nowych klientów.

```

1 import socket
2 import threading, queue
3
4 clients = {}
5 lock = threading.Lock()
6
7 # ... w tej chwili nieistotny kod ...
8
9 if __name__ == '__main__':
10     if len(sys.argv) != 2:
11         sys.stderr.write("usage: chat_server port\n")
12         exit(1)
13
14     try:
15         port = int(sys.argv[1])
16         assert port > 0
17     except:
18         sys.stderr.write("error: invalid port\n")
19         exit(1)
20
21     listen_sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
22     listen_sock.bind(("0.0.0.0", port))
23     listen_sock.listen(5)
24

```

```

25|     addr = listen_sock.getsockname()
26|     print('Listening on {}'.format(addr))
27|
28|     while True:
29|         client_sock,addr = listen_sock.accept()
30|         q = queue.Queue()
31|         with lock:
32|             clients[client_sock.fileno()] = {
33|                 'name': None,
34|                 'queue': q
35|             }
36|             recv_thread = threading.Thread(target=handle_client_recv,
37|                 args=[client_sock, addr], daemon=True)
38|             send_thread = threading.Thread(target=handle_client_send,
39|                 args=[client_sock, q, addr], daemon=True)
40|             recv_thread.start()
41|             send_thread.start()
42|             print('Connection from {}'.format(addr))

```

Listing 5.4: Odbieranie wiadomości. Źródło: [7].

Po pierwsze importujemy dwa nowe moduły. Pierwszy threading zawiera obiekty i funkcje służące do obsługi wielowątkowości, zaś drugi queue zawiera kolejkę FIFO, która jest „thread safe”. Co to znaczy? Okaże się za chwilę.

Na początku programu ustalamy dwie zmienne `clients` oraz `lock`. Zmienne te są współdzielone przez wszystkie wątki aplikacji i będzie można się do nich bezpośrednio odwoływać w wątkach. Na początku serwer tworzy gniazdo TCP i nasłuchuje na połaczenie, które pojawia się w linii 29. Następnie tworzy obiekt `Queue` i zapisuje w zmiennej `q`.

Obiekt `Queue` to kolejka FIFO (ang. First-In-First-Out), która jest przydatna do przesyłania danych między wątkami. Jej podstawową własnością jest to, że jest „thread safe”, czyli wszystkie wykonywane na niej operacje są atomowe, a dostęp do niej jest synchroniczny. Sama kolejka ma dwie metody, które będą nas szczególnie interesować, czyli `get` (pobranie najstarszego elementu kolejki) i `put` (dodanie nowego elementu do kolejki). Dzięki temu, że operacje są atomowe, dwa wątki nie są w stanie wywołać jednocześnie metod na obiekcie `Queue`. Dopiero, gdy jedna metoda się zakończy (np. dodawanie nowego elementu) to druga będzie mogła się wykonać (np. pobranie elementu przez inny wątek). W przeciwnym razie inny wątek będzie czekał, dopóki poprzedni wątek nie zakończy swojej operacji na obiekcie. Jeśli nie byłoby takiego mechanizmu, to zawartość kolejki mogłaby zostać niepoprawnie zmieniona, gdy np. dwa wątki jednocześnie chcieliby dodać obiekt do kolejki. Wywołanie metody `put` składa się z wielu instrukcji procesora i jeśli instrukcje z dwóch wątków zostały wykonane według szczególnego przelotu, to mogłyby się okazać, że mimo że oba wątki poprawnie wykonały metodę `put`, to w kolejce pojawił się tylko jeden obiekt.

Zatem stworzony i zapisany w zmiennej `q` obiekt `Queue` będzie służył jako bufor otrzymanych wiadomości od pozostałych klientów, które za pomocą kolejki będą przesyłane do klienta, który się właśnie połączył. Każdy klient posiada własną kolejkę.

Dalej w liniach 31-35 zapisywany jest w zmiennej `clients` słownik opisujący nowego klienta. Klient się jeszcze nie przedstawił (ma to zrobić w pierwszej

wiadomości), więc w polu name zapisujemy None. W polu queue zapisujemy wcześniej stworzony obiekt kolejki. Słownik natomiast jest zapisywany poid indeksem zwróconym przez metodę fileno na gnieździe klienta. Metoda ta zwraca deskryptor pliku skojarzonego z gniazdem, który jest liczbą naturalną. Istotne jest, że deskryptor ma wartość unikalną względem wszystkich gniazd, więc żadni dwaj podłączenie równocześnie klienci nie będą mieli takiego samego deskryptora pliku.

Cały proces zapisywania nowego słownika w zmiennej clients znajduje się w sekcji `with lock:` (linia 31), gdzie lock to współdzielona zmienna, w której znajduje się obiekt Lock z modułu `threading`. Obiekty Lock służą do zapewnienia tych samych funkcji, które dostarcza kolejka Queue, czyli atomowości oraz synchronizacji. W sekcji modyfikujemy współdzieloną zmienną `clients`, która jest słownikiem, a słowniki nie są obiektami typu „thread-safe”. Dlatego korzystamy z niezależnego obiektu Lock i jego działanie polega na tym, że jeśli jakiś wątek wejdzie do sekcji rozpoczynającej się `with lock:` (takich sekcji będzie więcej w programie), czyli zablokuje zmienną `lock`, to żaden inny wątek nie wejdzie do sekcji zaczynającej się tak samo, czyli blokującej tę samą zmienną `lock`. W przeciwnym razie mogłaby wystąpić sytuacja analogiczna do opisywanej przy okazji kolejki, czyli dwa wątki chciałby jednocześnie zmienić zawartość zmiennej `clients` i zmiany tylko jednego z nich zostałyby zapisane. Oczywiście można stworzyć kilka zmiennych typu Lock i za ich pomocą kontrolować dostęp do różnych zmiennych współdzielonych, które są od siebie niezależne.

W liniach 36-39 tworzymy dwa wątki. Jak widać, z tym celu tworzymy obiekty klasy Thread z modułu `threading`, która przyjmuje w konstruktorze: parametr `target`, który jest funkcją uruchamianą w oddzielnym wątku; parametr `args`, który jest listą parametrów przekazanych do funkcji z parametrem `target` w momencie uruchamiania wątku oraz parametr `daemon` z wartością `True` (domyślna wartość to `False`). Ostatni parametr oznacza, że program może zakończyć działanie, gdy taki wątek działa w tle. U nas taka sytuacja co prawda nie wystąpi, ponieważ wątek główny wciąż oczekuje na nowych klientów. Jednakże, gdy wciśniemy `Ctrl+C`, to program będzie mógł zakończyć działanie, bez potrzeby czekania na zakończenie działania wątków oznaczonych jako demony. Czyli w powyższym przykładzie, w wątkach zostaną uruchomione funkcje `handle_client_recv(client_sock, addr)` oraz `handle_client_send(client_sock, q, addr)`. Na końcu serwer uruchamia oba wątki i wyświetla informację o połączeniu.

Przejdzmy teraz do funkcji `handle_client_recv` oraz `handle_client_send`, które są uruchamiane w osobnych wątkach. Ich kod znajduje się poniżej:

```

1 def handle_client_recv(sock, addr):
2     """ Receive messages from client and broadcast them to
3         other clients until client disconnects """
4
5     rest = bytes()
6     while True:
7         try:
8             (msgs, rest) = recv_msg(sock, rest)
9         except (EOFError, ConnectionError):

```

```

10     handle_disconnect(sock, addr)
11     break
12 for msg in msgs:
13     print(' {}: {}'.format(addr, msg))
14
15     # Handle first message. Set clients name.
16     with lock:
17         client_name = clients[sock.fileno()]['name']
18         if client_name is None:
19             clients[sock.fileno()]['name'] = msg
20         else:
21             """ Add message to each connected client's send queue """
22             msg = '{}: {}'.format(client_name, msg)
23             for i in clients:
24                 clients[i]['queue'].put(msg)
25
26 def handle_client_send(sock, q, addr):
27     """ Monitor queue for new messages, send them to client as
28     they arrive """
29
30     while True:
31         msg = q.get()
32         if msg == None: break
33         try:
34             send_msg(sock, msg)
35         except (ConnectionError, BrokenPipe):
36             handle_disconnect(sock, addr)
37             break
38
39
40 def handle_disconnect(sock, addr):
41     """ Ensure queue is cleaned up and socket closed when a client
42     disconnects """
43
44     fd = sock.fileno()
45     with lock:
46         # Get send queue for this client
47         q = clients.get(fd, None)['queue']
48         # If we find a queue then this disconnect has not yet
49         # been handled
50         if q:
51             q.put(None)
52             del clients[fd]
53
54     addr = sock.getpeername()
55     print('Client {} disconnected'.format(addr))
56     sock.close()

```

Listing 5.5: Wątki obsługi klienta. Źródło: [7].

Pierwszą funkcją oddelegowaną do wątku jest `handle_client_recv`. Jej zadaniem jest odbieranie wiadomości od klienta i przekazywanie do wszystkich klientów. Musimy pamiętać jednak, że zgodnie z założeniami pierwsza wiadomość od klienta to jego nazwa, zaś do przekazywanych wiadomości doklejamy nazwę nadawcy. Funkcja działa w pętli nieskończonej (linia 6), ponieważ nie wiemy ile wiadomości będzie przesyłał klient. Pierwszym krokiem jest oczekiwanie na wiadomości oraz ich pobranie i zdekodowanie. Robimy to w linii 8 za pomocą wcześniej omówionej funkcji `recv_msgs`, której przekazujemy gniazdo klienta oraz zmienną `rest`, początkowo pustą. Jako, że komunikacja może zakończyć się wyjątkiem, to w razie wystąpienia przechwytyujemy go, kończymy połączenie za pomocą funkcji `handle_disconnect` i przerywamy pętlę.

Dalej każda wiadomość jest wyświetlana na konsoli serwera (linia 13). Ko-

lejny krok to obsługa pierwszej wiadomości od klienta. Sprawdzamy to porównując jego nazwę z wartością `None` (linia 18). Jeśli wartość `name` jest `None` to oznacza, że jeszcze żadna wiadomość nie została przesłana. Wartość pierwszej wiadomości zapisujemy w polu `name` w słowniku skojarzonym z klientem (linia 19). Jeśli wartość zmiennej `client_name` nie jest `None`, to znaczy, że wcześniej wysłano wiadomość ustalającą nazwę klienta i aktualna wiadomość powinna być przesłana do wszystkich klientów. Zanim to zrobimy, w linii 22 dodawana jest nazwa nadawcy do wiadomości. Tak przygotowaną wiadomość można już rozesłać, czyli dodać do kolejki wiadomości skojarzonej z każdym z klientów, którzy zapisani są we współdzielonej zmiennej `client` (linie 23-24).

Druga funkcja oddelegowana do wątku to `handle_client_send`, której zadaniem jest monitorowanie zawartości kolejki klienta i w momencie, gdy pojawi się w niej nowa wiadomość, funkcja wysyła ją do klienta. Ta funkcja również działa w pętli nieskończonej (linia 30). W kolejnej linii wywołuje metodę `get` na obiekcie kolejki klienta. Jeśli kolejka jest pusta, to metoda będzie czekać aż coś się pojawi. W linii 32 sprawdzamy, czy pobrana wartość to `None` i jeśli tak, to przerywamy pętlę, czyli kończymy działanie funkcji i wątku. Jest to sposób jak zakończenie wątku, w sytuacji, gdy klient zakończy połączenie (wykorzystamy ten pomysł w funkcji `handle_disconnect`). Po pobraniu wiadomości z kolejki jest ona wysyłana w linii 34 za pomocą funkcji `send_msg` (jeszcze ją zdefiniujemy). W sytuacji, gdy wystąpi jakiś wyjątek oznaczający błąd połączenia lub dostępu do kolejki, wywołujemy funkcję `handle_disconnect` i przerywamy pętlę.

Zadaniem ostatniej funkcji, czyli `handle_disconnect` jest zakończenie połączenia z klientem oraz usunięcie go ze zmiennej `clients`, żebyśmy nie próbowali wysyłać wiadomości do kogoś, kogo już nie ma. Funkcja zapisuje w zmiennej `fd` unikalny deskryptor pliku klienta. Następnie blokuje dostęp do współdzielonej zmiennej (linia 45) i w sekcji `with` wysyła do kolejki wartość `None`, dzięki której wątek obsługujący kolejkę zakończy działanie (patrz paragraf wyżej), usuwa słownik klienta ze zmiennej `clients` (linia 52), informuje o zamknięciu połączenia z klientem (linie 54-55) oraz zamyka gniazdo (linia 56).

Ostatnia funkcja, która została do omówienia to `send_msg`. Jej zadaniem jest dodanie znaku końca wiadomości o zakodowanie jej w formacie UTF-8.

```

1 def send_msg(sock, msg):
2     """ Send a string over a socket, preparing it first """
3     msg += '\0'
4     data = msg.encode('utf-8')
5     sock.sendall(data)

```

Listing 5.6: Wysłanie wiadomości. Źródło: [7].

W ten sposób omówiliśmy cały kod wielowątkowego serwera Chat. Teraz pora na klienta.

5.1.4 Klient Chat

Na pierwszy rzut oka, klient chat nie musi korzystać z wielowątkowości, bo przecież komunikuje się tylko z serwerem, a nie z wieloma klientami, jak serwer.

Jednakże klient musi jednocześnie odbierać wiadomości od serwera oraz czekać na wiadomość użytkownika. Dlatego klient również będzie wielowątkowy.

Klient, podobnie jak serwer, korzysta z funkcji `send_msg`, `parse_recdv_data` oraz `recv_msg`, które są takie same jak w przypadku serwera, dlatego pominiemy je.

```

1 sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
2 sock.connect((addr, port))
3 print('Connected to {}:{}'.format(addr, port))
4
5 # Create thread for handling user input and message sending
6 thread = threading.Thread(target=handle_input, args=[sock], daemon=True)
7 thread.start()
8
9 rest = bytes()
10 addr = sock.getsockname()
11 # Loop indefinitely to receive messages from server
12 while True:
13     try:
14         (msgs, rest) = recv_msg(sock, rest)
15         for msg in msgs:
16             print(msg)
17     except ConnectionError:
18         print('Connection to server closed')
19         sock.close()
20         break

```

Listing 5.7: Główny wątek klienta. Źródło: [7].

W głównym wątku program waliduje dane wejściowe (ip oraz port), po czym tworzy gniazdo i łączy się z serwerem (linie 1-2). Następnie w wątku w trybie demona uruchamia funkcję `handle_input`, przekazując jej gniazdo jako parametr. Funkcja ta będzie odpowiedzialna za przyjęcie wiadomości od użytkownika i wysłanie do serwera, ale do niej wróćmy za chwilę.

Dalej wątek główny uruchamia pętlę nieskończoną, w której za pomocą funkcji `recv_msg` (linia 14) pobiera wiadomości od serwera w taki sam sposób, w jaki serwer odbierał wiadomości od klienta. Każda wiadomość jest następnie wyświetlana na ekranie. Oczywiście cała ta operacja jest umieszczona w sekcji `try-catch`, żeby w przypadku zerwania połączenia i tym samym wystąpienia wyjątku `ConnectionError` klient zamknął połączenie i zakończył pętlę.

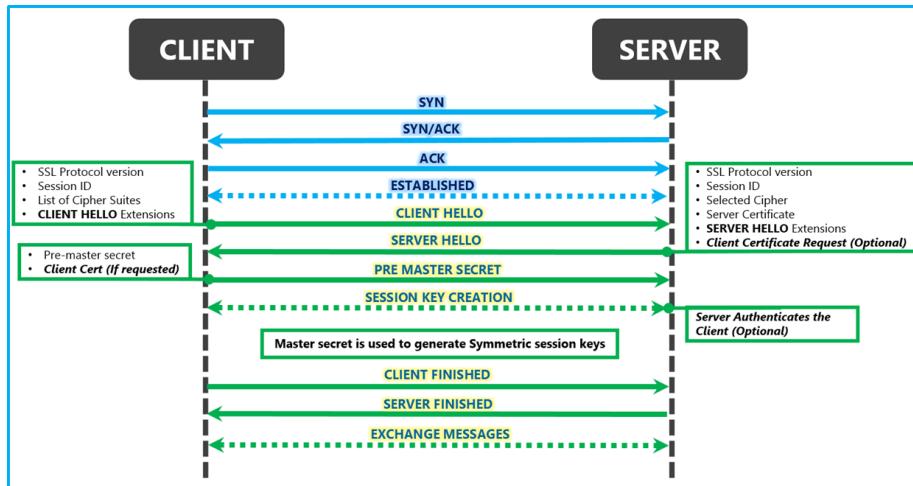
Przejdźmy teraz do funkcji `handle_input` uruchomionej w drugim wątku.

```

1 def handle_input(sock):
2     """ Prompt user for message and send it to server """
3     print("Type messages, enter to send. 'q' to quit")
4     while True:
5         msg = input()
6         if msg == 'q':
7             sock.shutdown(socket.SHUT_RDWR)
8             sock.close()
9             break
10    try:
11        send_msg(sock, msg)
12    except (BrokenPipeError, ConnectionError):
13        break

```

Listing 5.8: Wątek dla użytkownika. Źródło: [7].



Rysunek 5.3: TLS - diagram.
Źródło: blogs.msdn.microsoft.com/kaushal

Funkcja na początku wyświetla informację dla użytkownika, po czym uruchamia pętlę nieskończoną. W pętli czeka na wiadomość od użytkownika (linia 5). Skrypt sprawdza, czy wiadomość od użytkownika to „q”, które oznacza zakończenie działania. Jeśli użytkownik wpisał „q”, to połączenie oraz pętla są końcowe. W przeciwnym razie uruchamiana jest funkcja `send_msg`, omówiona przy okazji serwera, która wysyła wiadomość użytkownika do serwera.

Kod wielowątkowego serwera i klienta chat znajduje się w repozytorium w folderze `chat/multithreaded`. Oba skrypty powinny zostać uruchomione za pomocą interpretera języka Python w wersji 3.

5.2 Bezpieczne gniazda TLS/SSL

W tym rozdziale przyjrzymy się dokładniej szyfrowanym połączeniom, a konkretniej szyfrowanym protokołom SSL (ang. Secure Socket Layer) i TLS (ang. Transport Layer Security), które są wykorzystywane przez protokoły warstw wyższych.

TLS to standard będący rozwinięciem protokołu SSL, zaprojektowanego pierwotnie przez Netscape Communications. TLS zapewnia poufność i integralność transmisji danych, a także uwierzytelnienie serwera, a niekiedy również klienta. Opiera się na szyfrowaniu asymetrycznym oraz certyfikatach X.509.

Według modelu OSI, TLS działa w warstwie prezentacji, dzięki czemu może zabezpieczać protokoły warstwy najwyższej – warstwy aplikacji. W tym rozdziale wykorzystamy TLS, żeby dodać szyfrowanie do protokołu. W tym celu

Time	Source	Destination	Protocol	Length	Info
334 8.520515000	192.168.1.111	216.58.209.46	TCP	66	54324->443 [ACK] Seq=1 Ack=1 Win=293
335 8.520813000	192.168.1.111	216.58.209.46	TLSv1.2	583	Client Hello
336 8.526083000	216.58.209.46	192.168.1.111	TCP	66	443->54324 [ACK] Seq=1 Ack=518 Win=4
337 8.526206000	216.58.209.46	192.168.1.111	TLSv1.2	226	Server Hello, Change Cipher Spec, H
338 8.526228000	192.168.1.111	216.58.209.46	TCP	66	54324->443 [ACK] Seq=518 Ack=161 Win
339 8.526672000	192.168.1.111	216.58.209.46	TLSv1.2	282	Change Cipher Spec, Hello Request,
340 8.529338000	192.168.1.111	216.58.209.46	TLSv1.2	119	Application Data
341 8.529775000	192.168.1.111	216.58.209.46	TLSv1.2	158	Application Data, Application Data
342 8.530169000	192.168.1.111	216.58.209.46	TLSv1.2	934	Application Data, Application Data
343 8.532053000	216.58.209.46	192.168.1.111	TLSv1.2	122	Application Data

Handshake Protocol: Client Hello
 Handshake Type: Client Hello (1)
 Length: 508
 Version: TLS 1.2 (0x0303)
 Random
 Session ID Length: 32
 Session ID: eef48760aa1190bb2cd4f3fd4be2a015e6e0d6366fbe6f6...
 Cipher Suites Length: 34
 Cipher Suites (17 suites)
 Cipher Suite: TLS_ECDHE_ECDSA WITH AES_128_GCM_SHA256 (0xc02b)
 Cipher Suite: TLS_ECDHE_RSA WITH AES_128_GCM_SHA256 (0xc02f)
 Cipher Suite: TLS_ECDHE_ECDSA WITH AES_256_GCM_SHA384 (0xc02c)
 Cipher Suite: TLS_ECDHE_RSA WITH AES_256_GCM_SHA384 (0xc030)
 Cipher Suite: Unknown (0xccaa)
 Cipher Suite: Unknown (0xccab)
 Cipher Suite: TLS_ECDH_ECDSA WITH CHACHA20_POLY1305_SHA256 (0xc0cc)

Rysunek 5.4: TLS - Wireshark.

wykorzystamy moduł `ssl` dostępny wraz z Pythonem (nie będziemy się zagłębiać w techniczne aspekty protokołu TLS).

Na Rysunku 5.3 znajduje się schemat wymiany pakietów między klientem i serwerem w protokole HTTPS. Na niebiesko zaznaczone są pakiety z protokołu TCP, ponieważ TLS z niego korzysta i udostępnia bezpieczne połączenie dla protokołów z warstwy aplikacji. Następnie klient wysyła pakiet `Client Hello`, w którym informuje serwer m.in. o wspieranym protokole TLS, czy zestawie szyfrów, które wspiera. W odpowiedzi serwer wysyła pakiet `Server Hello`, w którym informuje o swojej wersji protokołu, o wybranym szyfrze, o swoim certyfikacie oraz opcjonalnie o dodatkowych rozszerzeniach. Opcjonalnie serwer może poprosić o certyfikat klienta. Klient, korzystając z certyfikatu serwera, przygotowuje i odsyła zaszyfrowany kluczem publicznym serwera sekret (`Pre Master Secret`), z którego generowany jest sesyjny klucz symetryczny. Po wygenerowaniu klucza klient i serwer wymieniają się wiadomościami `Client Finished` i `Server Finished` w celu zakończenia fazy nawiązywania zaszyfrowanego połączenia. Od tej chwili może wskroczyć protokół z warstwy wyższej i wszystkie dane przesyłane są zaszyfrowane kluczem sesyjnym.

5.2.1 Klient HTTPS

Pierwszym przykładem będzie klient HTTPS, czyli klient pobierający zasoby przez zaszyfrowany protokół HTTP. Oprócz funkcji zwykłego klienta HTTP, ten skrypt będzie dodatkowo sprawdzał poprawność certyfikatu serwera WWW.

```

1 import socket
2 import ssl
3 from ssl import wrap_socket, CERT_NONE, PROTOCOL_TLSv1_2, SSLError
4 from ssl import SSLContext
5 from ssl import HAS_SNI

```

```

6  from pprint import pprint
7
8 TARGET_HOST = 'www.google.com'
9 SSL_PORT = 443
10 # Use the path of CA certificate file in your system
11 CA_CERT_PATH = './GeoTrustGlobalCA.pem'
12
13 def ssl_wrap_socket(sock, keyfile=None, certfile=None,
14 cert_reqs=None, ca_certs=None, server_hostname=None,
15 ssl_version=None):
16
17     context = SSLContext(ssl_version)
18     context.verify_mode = cert_reqs
19
20     if ca_certs:
21         try:
22             context.load_verify_locations(ca_certs)
23         except Exception as e:
24             raise SSLError(e)
25
26     if certfile:
27         context.load_cert_chain(certfile, keyfile)
28
29     if HAS_SNI: # OpenSSL enabled SNI
30         return context.wrap_socket(sock, server_hostname=server_hostname)
31
32     return context.wrap_socket(sock)
33
34 if __name__ == '__main__':
35     hostname = input("Enter target host:") or TARGET_HOST
36     client_sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
37     client_sock.connect((hostname, 443))
38
39     ssl_socket = ssl_wrap_socket(client_sock, ssl_version=PROTOCOL_TLSv1_2,
40         cert_reqs=ssl.CERT_REQUIRED, ca_certs=CA_CERT_PATH, server_hostname=hostname)
41
42     print("Extracting remote host certificate details:")
43     cert = ssl_socket.getpeercert()
44     pprint(cert)
45     if not cert or ('commonName', TARGET_HOST) not in cert['subject'][4]:
46         raise Exception("Invalid SSL cert for host %s. Check if this is a man-in-the-
47                         middle attack!")
48
49     ssl_socket.write('GET / \n'.encode('utf-8'))
50     pprint(ssl_socket.recv(1024).split(b"\r\n"))
51     ssl_socket.close()
52     client_sock.close()

```

Listing 5.9: Klient HTTPS (źródło: [7])

W liniach 34-37 tworzymy zwykłe gniazdo TCP. Dalej w linii 39 wywołujemy funkcję `ssl_wrap_socket`, który zamienia nasz zwykły socket (przekazany jako pierwszy parametr) i zwraca gniazdo, w którym komunikacja jest szyfrowana (zaraz wróćmy do parametrów). Dalej w liniach 48 i 49 przesyłamy i odbieramy dane tak, jakbyśmy to robili ze zwykłym gniazdem. Na koncu w liniach 50 i 51 zamykane są odpowiednio: gniazdo zabezpieczone i gniazdo zwykłe, opakowane zabezpieczonym.

W powyższym opisie pominęliśmy funkcję `ssl_wrap_socket`. Przyjmuje ona w parametrach następujące dane:

- `sock` - gniazdo TCP,
- `keyfile` - ścieżka do pliku z prywatnym kluczem; nie podajemy jej, ponieważ

w naszym przykładzie serwer nie weryfikuje tożsamości klienta,

- certfile - ścieżka do pliku z certyfikatem klienta; nie podajemy jej, ponieważ w naszym przykładzie serwer nie weryfikuje tożsamości klienta,
- cert_reqs - tryb weryfikacji serwera, w którym określamy, czy weryfikacja jest wymagana, opcjonalna lub jej nie ma; podajemy wartość CERT_REQUIRED co oznacza, że zabezpieczone gniazdo musi zweryfikować certyfikat serwera,
- ca_certs - ścieżka do pliku z zaufanymi certyfikatami; certyfikat zostanie uznany za ważny, gdy będzie wystawiony przez jeden z certyfikatów zaufanych; podajemy ścieżkę do pliku z jednym certyfikatem zaufanym (*GeoTrust Inc.*),
- server_hostname - domena, dla której weryfikujemy certyfikat; podajemy domenę, z którą będziemy się łączyć,
- ssl_version - wersja protokołu SSL/TLS, z której chce skorzystać klient; podajemy wartość PROTOCOL_TLSv1_2, która odpowiada protokołowi TLS w wersji 1.2.

Funkcja `ssl_wrap_socket` tworzy obiekt `SSLContext`, któremu przekazuje wersję protokołu w parametrze. Obiekt ten nie reprezentuje bezpośrednio połączenia TLS, ponieważ zawiera dane, których czas życia jest dłuższy niż czas życia połączenia, np. certyfikaty, klucze, itp. Dla uproszenia jednak możemy przyjąć, że jest to połączenie SSL/TLS. W kontekście ustawiany jest tryb weryfikacji w linii 18. Następnie, jeśli podano ścieżkę do zaufanych certyfikatów, są one wczytywane w linii 22. Operacja ta może zakończyć się błędem (np. format certyfikatu będzie niepoprawny), który będzie przechycony i zamieniony na błąd `SSLError` w linii 24. Następnie, jeśli podano ścieżki do certyfikatu i lucza prywatnego klienta, są one wczytywane w linii 27. W naszym przypadku operacja ta nie jest wywoływana, ponieważ serwer WWW, z którym się łączymy (`www.google.com`) nie wymaga posiadania certyfikatu. Inaczej musielibyśmy wyupić certyfikat, żeby korzystać z wyszukiwarki Google.

W linii 29 sprawdzamy, czy ustawione jest pole `HAS_SNI` w module `ssl`, które oznacza, czy biblioteka OpenSSL, z której korzystamy wspiera rozszerzenie Server Name Indication (opisane w RFC 4366). Dzięki niemu klient może wysłać do serwera informację o domenie, z którą się chce połączyć w ramach pakietu Client Hello. Jeśli OpenSSL wspiera SNI, to zwracamy wynik metody `wrap_socket` wywołanej na wcześniej stworzonym obiekcie `context`, przekazując w parametrze zwykłe gniazdo oraz domenę w parametrze `server_hostname`. W przeciwnym razie zwracamy wynik tej samej metody, jednakże bez parametru `server_hostname`.

Oprócz komunikacji między klientem, a serwerem WWW klient sprawdza w liniach 42-46 informacje o certyfikacie serwera. Pobiera je za pomocą metody `getpeercert` i wyświetla na ekranie. W linii 45 klient sprawdza, czy serwer

posiada caertyfikat oraz czy w podmiocie (pole `Subject`) pod indeksem 4 znajduje się domena, z którą chcemy się połączyć. W przeciwnym razie mielibyśmy do czynienia z podłożeniem fałszywego certyfikatu, czyli z klasycznym atakiem Man-in-the-Middle.

5.2.2 Szyfrowany czat

W poprzednim podrozdziale napisaliśmy klienta korzystającego z szyfrowanego połączenia i łączącego się z serwerem `www.google.com`. W tym podrozdziale rozwiniemy serwer oraz klienta chat, żeby korzystały z szyfrowanego połączenia.

```

1 import sys
2 import socket
3 import threading, queue
4 import ssl
5
6 clients = {}
7 lock = threading.Lock()
8
9
10 def parse_recv_data(data):
11     """ Break up raw received data into messages, delimited
12         by null byte """
13     parts = data.split(b'\0')
14     msgs = parts[:-1]
15     rest = parts[-1]
16     return (msgs, rest)
17
18
19 def recv_msg(sock, data=b''):
20     """ Receive data and break into complete messages on null byte
21         delimiter. Block until at least one message received, then
22         return received messages """
23
24     msgs = []
25     while not msgs:
26         recvdata = sock.recv(4096)
27         if not recvdata:
28             raise ConnectionError()
29         data += recvdata
30         (msgs, rest) = parse_recv_data(data)
31     msgs = [msg.decode('utf-8') for msg in msgs]
32     return (msgs, rest)
33
34
35 def send_msg(sock, msg):
36     """ Send a string over a socket, preparing it first """
37     msg += '\0'
38     data = msg.encode('utf-8')
39     sock.sendall(data)
40
41
42 def handle_client_recv(ssl_sock, sock, addr):
43     """ Receive messages from client and broadcast them to
44         other clients until client disconnects """
45
46     rest = bytes()
47     while True:
48         try:
49             (msgs, rest) = recv_msg(ssl_sock, rest)
50         except (EOFError, ConnectionError):
51             handle_disconnect(ssl_sock, sock, addr)
52             break
53         for msg in msgs:

```

```

54     print('{}: {}'.format(addr, msg))
55
56     # Handle first message. Set clients name.
57     with lock:
58         client_name = clients[ssl_sock.fileno()]['name']
59         if client_name is None:
60             clients[ssl_sock.fileno()]['name'] = msg
61         else:
62             """ Add message to each connected client's send queue """
63             msg = '{}: {}'.format(client_name, msg)
64             for i in clients:
65                 clients[i]['queue'].put(msg)
66
67     def handle_client_send(ssl_sock, sock, q, addr):
68         """ Monitor queue for new messages, send them to client as
69             they arrive """
70
71         while True:
72             msg = q.get()
73             if msg == None: break
74             try:
75                 send_msg(ssl_sock, msg)
76             except (ConnectionError, BrokenPipe):
77                 handle_disconnect(ssl_sock, sock, addr)
78             break
79
80
81     def handle_disconnect(ssl_sock, sock, addr):
82         """ Ensure queue is cleaned up and socket closed when a client
83             disconnects """
84
85         fd = ssl_sock.fileno()
86         with lock:
87             # Get send queue for this client
88             q = clients.get(fd, None)['queue']
89             # If we find a queue then this disconnect has not yet
90             # been handled
91             if q:
92                 q.put(None)
93                 del clients[fd]
94
95             addr = ssl_sock.getpeername()
96             print('Client {} disconnected'.format(addr))
97             ssl_sock.close()
98             sock.close()
99
100
101 if __name__ == '__main__':
102     if len(sys.argv) != 2:
103         sys.stderr.write("usage: chat_server port\n")
104         exit(1)
105
106     try:
107         port = int(sys.argv[1])
108         assert port > 0
109     except:
110         sys.stderr.write("error: invalid port\n")
111         exit(1)
112
113     listen_sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
114     listen_sock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
115     listen_sock.bind(("0.0.0.0", port))
116     listen_sock.listen(5)
117
118     addr = listen_sock.getsockname()
119     print('Listening on {}'.format(addr))
120
121     while True:

```

```

122     client_sock,addr = listen_sock.accept()
123     ssl_client = ssl.wrap_socket(client_sock, server_side=True,
124         certfile="server.crt", keyfile="server.key",
125         ssl_version=ssl.PROTOCOL_TLSv1_2)
126
127     q = queue.Queue()
128     with lock:
129         clients[ssl_client.fileno()] = {
130             'name': None,
131             'queue': q
132         }
133     recv_thread = threading.Thread(target=handle_client_recv,
134         args=[ssl_client, client_sock, addr], daemon=True)
135     send_thread = threading.Thread(target=handle_client_send,
136         args=[ssl_client, client_sock, q, addr], daemon=True)
137     recv_thread.start()
138     send_thread.start()
139     print('Connection from {}'.format(addr))

```

Listing 5.10: Serwer TLS

Powyższy kod serwera jest taki sam, jak ten omawiany w podrozdziale o wielowątkowości z tą różnicą, że wykorzystuje szyfrowane połączenie. W tej chwili omówimy jedynie różnice względem wersji nieszyfrowanej.

Serwer tworzy zwykłe, nasłuchujące gniazdo w liniach 113-116. Gniazdo musi korzystać z protokołu TCP, ponieważ inne nie są wspierane przez SSL/TLS. Następnie w nieskończonej pętli, w linii 122 przyjmuje połączenie TCP. Do tej pory wszystko jest takie samo, jak w zwykłym serwerze chatu. W linii 123 wywoływana jest funkcja `wrap_socket` z modułu `ssl`, która przyjmuje jako parametry:

- gniazdo TCP;
- `server_side` ustawiony na `True`, co znacza, że to gniazdo pełni rolę serwera (istotne podczas nawiązywania szyfrowanego połączenia);
- w `certfile` ścieżkę do certyfikatu, którym będzie się przedstawiał serwer,
- w `keyfile` ścieżkę do klucza prywatnego powiązanego z certyfikatem,
- w `ssl_version` wersje protokołu (TLS 1.2).

Od tej chwili w zmiennej `ssl_client` jest gniazdo korzystające z szyfrowanego połączenia. Podobnie jak w zwykłym serwerze chatu, w dalszej kolejności uruchamiane są wątki obsługujące klienta, które pobierają od niego wiadomości i wysyłają do niego wiadomości. Różnica polega na tym, że do listy argumentów dodawane jest szyfrowane gniazdo `ssl_client`. Jest ono wykorzystywane do pobierania i wysyłania danych. Gniazdo nieszyfrowane jest wykorzystywane jedynie w funkcji `handle_disconnect` w liniach 97-98. Najpierw zamykane jest gniazdo szyfrowane, które tak właściwie jest opakowaniem gniazda zwykłego. Następnie zamykane jest gniazdo TCP (`sock`).

```

1 import sys
2 import socket
3 import threading
4 import ssl

```

```

5
6
7 def send_msg(sock, msg):
8     """ Send a string over a socket, preparing it first """
9     msg += '\0'
10    data = msg.encode('utf-8')
11    sock.sendall(data)
12
13
14 def parse_recv_data(data):
15     """ Break up raw received data into messages, delimited
16     by null byte """
17     parts = data.split(b'\0')
18     msgs = parts[:-1]
19     rest = parts[-1]
20     return (msgs, rest)
21
22
23 def recv_msg(sock, data=b''):
24     """ Receive data and break into complete messages on null byte
25     delimiter. Block until at least one message received, then
26     return received messages """
27
28     msgs = []
29     while not msgs:
30         recvdata = sock.recv(4096)
31         if not recvdata:
32             raise ConnectionError()
33         data = data + recvdata
34         (msgs, rest) = parse_recv_data(data)
35         msgs = [msg.decode('utf-8') for msg in msgs]
36     return (msgs, rest)
37
38
39 def handle_input(ssl_sock, sock):
40     """ Prompt user for message and send it to server """
41     print("Type messages, enter to send. 'q' to quit")
42     while True:
43         msg = input() # Blocks
44         if msg == 'q':
45             ssl_sock.close()
46             sock.shutdown(socket.SHUT_RDWR)
47             sock.close()
48             break
49     try:
50         send_msg(ssl_sock, msg) # Blocks until sent
51     except (BrokenPipeError, ConnectionError):
52         break
53
54 if __name__ == "__main__":
55
56     if len(sys.argv) != 3:
57         sys.stderr.write("usage: chat_client ip port\n")
58         exit(1)
59
60     try:
61         addr = sys.argv[1]
62         port = int(sys.argv[2])
63         assert port > 0
64     except:
65         sys.stderr.write("error: invalid port\n")
66         exit(1)
67
68     sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
69     ssl_sock = ssl.wrap_socket(sock, cert_reqs=ssl.CERT_REQUIRED,
70                               ssl_version=ssl.PROTOCOL_TLSv1_2, ca_certs="trusted_certs.crt")
71     ssl_sock.connect((addr, port))
72     print('Connected to {}:{}'.format(addr, port))

```

```

73  # check remote cert
74  cert = ssl_conn.getpeercert()
75  print("Checking server certificate")
76  if not cert or ssl.match_hostname(cert, "PAS - TLS server"):
77      raise Exception("Invalid SSL cert.")
78  print("Server certificate OK.")
79
80
81  # Create thread for handling user input and message sending
82  thread = threading.Thread(target=handle_input, args=[ssl_sock, sock], daemon=
83      True)
84  thread.start()
85
86  rest = bytes()
87  addr = ssl_sock.getsockname()
88  # Loop indefinitely to receive messages from server
89  while True:
90      try:
91          # blocks
92          (msgs, rest) = recv_msg(ssl_sock, rest)
93          for msg in msgs:
94              print(msg)
95      except ConnectionError:
96          print('Connection to server closed')
97          sock.close()
98          break
99

```

Listing 5.11: Klient TLS

Klient tworzy szyfrowane gniazdo w linii 69, przed połączeniem z serwerem. W linii tej wywoływana jest funkcja `wrap_socket` z modulu `ssl`, która podobnie jak w przypadku serwera przyjmuje zwykłe gniazdo TCP jako pierwszy parametr oraz wersję SSL w `ssl_version`. Jednakże w przeciwnieństwie do serwera przyjmuje również parametr `cert_reqs`, w którym ustalamy, że klient wymaga certyfikatu od klienta oraz `ca_certs`, który jest ścieżką do pliku z zaufanymi certyfikatami. Następnie w linii 26 klient łączy się z serwerem za pomocą szyfrowanego połączenia (SSL Handshake). W liniach 74-79 pobierany i weryfikowany jest certyfikat serwera.

Od linii 81 klient może się komunikować z serwerem tak samo, jak nieszyfrowany klient chat. Różnica polega na tym, że szyfrowane gniazdo jest przekazywane w liście argumentów do wątku oraz na nim wywoływanie są funkcje pobierania i wysyłania wiadomości.

Na Rysunku 5.5 znajduje się fragment strumienia danych wymienianych między serwerem i klientem korzystającymi z szyfrowanego połączenia.

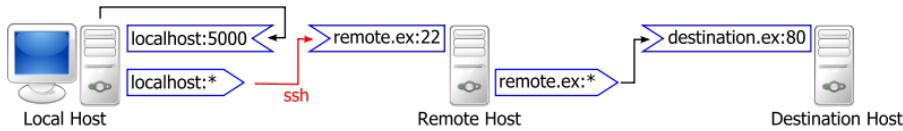
5.2.3 Tunelowanie SSH

Omawiając temat szyfrowanych połączeń warto omówić tunelowanie, które polega na przesyłaniu niezabezpieczonych pakietów protokołów TCP (POP3, SMTP czy HTTP) przez bezpieczny protokół SSH.

Zastosować tuneli SSH może być bardzo dużo, od prostego obejścia blokad serwisów WWW, po chęć zabezpieczenia jawnej komunikacji. Na przykład jeśli z naszej sieci mamy zablokowany dostęp do pewnych stron WWW, ale mamy dostęp do jakiegoś serwera SSH z poza naszej sieci, to możemy się połączyć z tym serwerem SSH, a on w naszym imieniu będzie łączył się z zablokowanymi

```
.....
.#...
.....>....>.4.....D-A*V.$.....\....0.....#.
..*H..
....0\1.0...U....PL1.0...U....Lubelskie1.0
..U....Lublin1
0...U.
..UMCS1.0...U....PAS - TLS server0..
170507190715Z.
180507190715Z0\1.0...U....PL1.0...U....Lubelskie1.0
..U....Lublin1
0...U.
..UMCS1.0...U....PAS - TLS server0.."0
..*H..
.....0..
.....eE.3.....e..>E....TD.....@N.P._.$C...&f.&....6....
..W..g1.Y..mpP.....
..LHrp.K..|..k.b...KC=.....5...!...../I?....<F..2..iy.....
..*H..
.....Z..."..K..0..7v.....^..=.49.0.....8s.."l.1*
..2..?U..[..D....n.....xAY.5u$.W...V..X...c9..kd.m
..&..M..O..T..?l^..).L...p.W.*..Uv.....L..-".x.
.....M..I..A.\.....,
.g..I}..Q.H.x.....Y..K.C.l6Lk..@.q=#V.VZx.G.....y9.....a
[H...../.....C%..(..w09.-..7.zh.).0H....e].f..D5.k....p..U..Y
....N.D.;.....FA.....F..BA..E..D..A.H.t8.
....y....x....S.....(....H/x.....(....)E.Yn.%...
.1....2k...rx...~@..R.0....m...M.r....?1Y:..(....{....e_.
{xIx...bY.h^.....)E.Z....p..8,<&
g.Eb.....].bLlx8A....P.....'s....|
```

Rysunek 5.5: TLS - strumień danych.



Rysunek 5.6: Diagram tunelu SSH. Źródło: Wikipedia.

do tej pory stronami. W tym celu można również skorzystać z serwera proxy, którego rolę tunel SSH również pełni, jednakże zwykle proxy nie zagwarantuje drugiej cechy tunelu SSH, czyli szyfrowania połączenia.

W przypadku poniższego skryptu wykorzystam wspomnianą możliwość zabezpieczenia komunikacji. Założmy, że mamy na poewnym serwerze uruchomiony serwer FTP oraz SSH. Serwer FTP wymaga autoryzacji, jednakże nie jest ona szyfrowana, co znaczy, że jeśli ktoś może podsłuchać naszą komunikację (co nie jest trudne, jeśli korzystamy z sieci Wifi) to mógłby podsłuchać nasz login i hasło do FTP. W celu zabezpieczenia się przed takim atakiem połączymy się z serwerem SSH (połączenie szyfrowane), a on przekaże połączenie do serwera FTP. Dzięki temu, że serwery SSH i FTP są na tej samej maszynie, to nasz login i hasło do serwera FTP nie będzie możliwe do podejrzenia. Natomiast login i hasło do serwera SSH będzie oczywiście zaszyfrowane.

Diagram tunelu SSH przedstawiam na Rysunku 5.6. Pierwszym krokiem jest nawiązanie połączenia między naszym serwerem (*Local Host*), a serwerem SSH (oznaczonym jako *Remote Host*) na porcie 22 (domyślnym porcie serwera

SSH). Następnie, serwer SSH otrzymuje instrukcję, aby wszystkie dane, które otrzyma od nas przekierował do serwera docelowego (oznaczonego *Destination Host*). Analogicznie, odpowiedź z serwera docelowego powinien przekazywać do nas. W naszym przykładzie serwer SSH i serwer docelowy (FTP) to będzie ta sama maszyna, przy czym serwer SSH będzie się łączył z serwerem FTP po interfejsie lokalnym (*loopback*). Tak zestawione połączenie między naszą maszyną (*Local Host*) i serwerem SSH (*Remote Host*) nasza maszyna wystawia do komunikacji na wybranym, wysokim porcie (oznaczone na diagramie sprzężeniem zwrotnym na porcie 5000). Od tej pory, łącząc się lokalnie z portem 5000 łączymy się z serwerem FTP za pośrednictwem połączenia po SSH, czyli nasze dane są szyfrowane, ponieważ wykorzystujemy SSH oraz dochodzą do serwera FTP, bo serwer SSH je przekazuje.

Jeśli korzystasz z kontenera Docker, to skrypty wykorzystywane do tunelowania SSH są w folderze *ssh_tunnel/*. W naszym przykładzie kontener Dockera będzie pełnił rolę serwera SSH i FTP (na połączenia FTP nasłuchuje tylko na interfejsie lokalnym). Stąd, żeby tunelowanie miało sens, sugeruję uruchomić skrypt serwera na własnej maszynie, a nie w kontenerze. Wtedy na Twojej maszynie na wybranym porcie wystawione zostanie szyfrowane połączenie z serwerem FTP.

Möżesz wykorzystać użytkownika `test` w kontenerze Dockera, którego hasło jest takie samo jak login, czyli `test`. Jeśli nie znasz adresu IP kontenera, to połącz się z nim i uruchom komendę `ifconfig`. Przy jednym z interfejsów powinien być widoczny adres IP. W moim przypadku jest to `172.17.0.2`.

Na listingu 5.12 znajduje się skrypt, który uruchamia połączenie z serwerem FTP, które jest szyfrowane, dzięki pośredniemu serwerowi SSH.

```

1 import sys
2 from sshtunnel import SSHTunnelForwarder
3 from getpass import getpass
4
5
6 if __name__ == "__main__":
7
8     if len(sys.argv) != 3:
9         sys.stderr.write("usage: %s user@ssh_server:ssh_port destination_server:\n"
10                         "destination_port\n" % \
11                         sys.argv[0])
12         exit(1)
13
14     try:
15         ssh_port = sys.argv[1].split(':')[1]
16         ssh_port = int(ssh_port)
17         assert ssh_port > 0
18     except:
19         sys.stderr.write("error: invalid ssh port\n")
20         exit(1)
21
22     try:
23         ssh_user = sys.argv[1].split('@')[0]
24         ssh_addr = sys.argv[1].split('@')[1].split(':')[0]
25     except:
26         sys.stderr.write("error: invalid ssh address\n")

```

```
damian@damian-dell:~/classes-pas/ssh_tunnels$ python ssh_tunnel.py test@172.17.0.2:22 localh
ost:21
Enter password for test@172.17.0.2:22:
Connected the remote service via local port: 34161
^CExiting user user request.

damian@damian-dell:~/classes-pas/ssh_tunnels$ ftp localhost 34161
Connected to localhost.
220 ProFTPD 1.3.5rc3 Server (Debian) [::1]
Name (localhost:damian): LOGIN
331 Password required for LOGIN
Password:
530 Login incorrect.
530 Login failed.
530 Remote system type is UNIX.
Using binary mode to transfer files.
ftp> 
```

Rysunek 5.7: Użycie tunelu SSH.

```
26    exit(1)
27
28 try:
29     destination_port = sys.argv[2].split(':')[1]
30     destination_port = int(destination_port)
31     assert destination_port > 0
32     destination_addr = sys.argv[2].split(':')[0]
33 except:
34     sys.stderr.write("error: invalid destination port\n")
35     exit(1)
36
37     ssh_password = getpass('Enter password for %s@%s:%s : ' % (ssh_user, ssh_addr,
38                         ssh_port))
39     server = SSHTunnelForwarder(ssh_address=(ssh_addr, ssh_port), ssh_username=
39                               ssh_user,
40                               ssh_password=ssh_password, remote_bind_address=(destination_addr,
41                                         destination_port))
40     server.start()
41
42     print('Connected the remote service via local port: %s' %server.local_bind_port)
43     try:
44         while True:
45             pass
46     except KeyboardInterrupt:
47         print("Exiting user user request.\n")
48     server.stop()
```

Listing 5.12: Tunel SSH

Skrypt wykorzystuje bibliotekę sshtunnel, którą można doinstalować za pomocą komendy pip install sshtunnel. Drugim, nowym pakietem jest getpass, który jest wbudowany i pozwala na pobranie od użytkownika tekstu bez jego wyświetlenia na ekranie. Jak sama nazwa wskazuje, przydatne do pobierania hasła.

W liniach 8-35 przeprowadzana jest walidacja oraz parsowanie argumentów programu. Program przyjmuje 2 argumenty, pierwszy to nazwa użytkownika, adres oraz port serwera SSH, zaś drugi to adres i port, z którymi ma być zestrojony tunel SSH (np. adres kontenera i port 21, czyli FTP). W linii 37 program pyta użytkownika o hasło do serwera SSH, zaś cała „magia” tworzenia tunelu i wystawiania go na lokalnym porcie w formie serwera odbywa się w liniach 38-40. Następnie, w nieskończonej pętli program oczekuje na zakończenie działania przez użytkownika (np. poprzez wcisnięcie Ctrl+C). Przez cały czas, przez który tunel ma być włączony program musi działać - stąd pętla nieskończona. Wyjątek KeyboardInterrupt przechwytuje wcisnięcie Ctrl+C, po którym serwer kończy działanie. Wcześniej, w linii 42 program wyświetla informację, na którym lokalnym porcie wystawiony jest tunel.

Na Rysunku 5.7 znajdują się dwa terminale. Po lewej stronie uruchomiony

1 0.0000000000 172.17.8.1	172.17.8.2	TCP	74 50128-22 [SYN] Seq=0 Win=29280 Len=0 MSS=1460 SACK_PERM=1 TStamp=1633761 TSecr=0 WS=128
2 0.0000072080 172.17.8.2	172.17.8.1	TCP	74 22-50128 [SYN ACK] Seq=1 Win=28960 Len=0 MSS=1460 SACK_PERM=1 TStamp=1633761 TSecr=0 WS=128
3 0.0001218000 172.17.8.1	172.17.8.2	TCP	66 50128-22 [ACK] Seq=1 Ack=1 Win=29312 Len=0 TStamp=1633761 TSecr=1633761
4 0.0011218000 172.17.8.1	172.17.8.2	SHTTP2	91 Client: Protocol [SSH-2.0-paramiko 1.16.0]
5 0.0011218000 172.17.8.2	172.17.8.1	TCP	66 50128-22 [ACK] Seq=1 Win=29056 Len=0 TStamp=1633761 TSecr=1633761
6 0.0160290000 172.17.8.2	172.17.8.1	SHTTP2	189 Server: Protocol [SSH-2.0-OpenSSH 6.8.1p1 Ubuntu/12.04]
7 0.0168768000 172.17.8.1	172.17.8.2	TCP	66 50128-22 [ACK] Seq=26 Ack=44 Win=29312 Len=0 TStamp=1633765 TSecr=1633765
8 0.0181969000 172.17.8.1	172.17.8.2	SHTTP2	666 Client: Key Exchange Init
9 0.0182599000 172.17.8.2	172.17.8.1	SHTTP2	1714 Server: Key Exchange Init
10 0.0183999000 172.17.8.1	172.17.8.2	TCP	66 50128-22 [ACK] Seq=626 Ack=1692 Win=32512 Len=0 TStamp=1633766 TSecr=1633766
11 0.0184399000 172.17.8.1	172.17.8.2	SHTTP2	238 Client: Diffie-Hellman Key Exchange Init
12 0.0381258000 172.17.8.2	172.17.8.1	SHTTP2	786 Server: Diffie-Hellman Key Exchange Reply, New Keys
13 0.0541298000 172.17.8.1	172.17.8.2	SHTTP2	82 Client: New Keys
14 0.0940688000 172.17.8.1	172.17.8.2	TCP	66 22-50128 [ACK] Seq=2412 Ack=786 Win=31360 Len=0 TStamp=1633785 TSecr=1633775
15 0.0941168000 172.17.8.1	172.17.8.2	SHTTP2	130 Client: Encrypted packet (len=64)
16 0.0941328000 172.17.8.2	172.17.8.1	TCP	66 22-50128 [ACK] Seq=2412 Ack=650 Win=31360 Len=0 TStamp=1633785 TSecr=1633785
17 0.0947610000 172.17.8.2	172.17.8.1	SHTTP2	130 Server: Encrypted packet (len=64)

Rysunek 5.8: Podgląd pakietów w tunelu SSH.

4 0.0022080000 172.17.8.2	8.8.8.8	DNS	83 Standard query 0x1a8c PTR 1.0.17.172.in-addr.arpa
5 0.0073550000 02:42:2a:70:6a:4c	02:42:ac:11:00:02	ARP	42 Who has 172.17.8.2? Tell 172.17.8.1
6 0.0073910000 02:42:2a:70:6a:4c	02:42:2a:70:6a:4c	ARP	42 Who has 172.17.8.1? Tell 172.17.8.2
7 5.0075440000 02:42:2a:70:6a:4c	02:42:ac:11:00:02	ARP	42 172.17.8.1 is at 02:42:2a:70:6a:4c
8 5.0075240000 02:42:2a:70:6a:4c	02:42:2a:70:6a:4c	ARP	42 172.17.8.2 is at 02:42:ac:11:00:02
9 5.0075240000 02:42:2a:70:6a:4c	02:42:2a:70:6a:4c	DNS	93 Standard query 0x1a8c PTR 1.0.17.172.in-addr.arpa
10 5.0473208000 8.8.4.4	172.17.8.2	DNS	83 Standard query Response 0x1a8c No such name
11 5.0493418000 172.17.8.2	172.17.8.1	FTP	124 Response: 220 ProFTPD 1.3.5rc3 Server (Debian) [:ffff:172.17.8.2]
12 5.0494950000 172.17.8.1	172.17.8.2	TCP	66 34096-21 [ACK] Seq=1 Win=29312 Len=0 TStamp=1668254 TSecr=1668254
13 8.1086590000 172.17.8.1	172.17.8.2	FTP	78 Request: USER LOGIN
14 8.1086680000 172.17.8.2	172.17.8.1	FTP	66 21-34496 [ACK] Seq=59 Ack=13 Win=29056 Len=0 TStamp=1669017 TSecr=1669017
15 8.1086680000 172.17.8.1	172.17.8.2	FTP	66 34496-21 [ACK] Seq=13 Win=29056 Len=0 TStamp=1669017 TSecr=1669017
16 8.1014640000 172.17.8.1	172.17.8.2	TCP	66 34096-21 [ACK] Seq=13 Win=29312 Len=0 TStamp=1669017 TSecr=1669017
17 11.1084290000 172.17.8.1	172.17.8.2	FTP	81 Request: PASS PASSWORD
18 11.1095910000 172.17.8.2	172.17.8.1	FTP	66 Response: 530 Login Incorrect.
19 11.1095760000 172.17.8.1	172.17.8.2	TCP	66 34096-21 [ACK] Seq=28 Ack=114 Win=29312 Len=0 TStamp=1669769 TSecr=1669769
20 11.1098150000 172.17.8.1	172.17.8.2	FTP	72 Request: SYST

Rysunek 5.9: Podgląd pakietów w połączeniu z serwerem FTP.

jest tunel, który łączy się z serwerem SSH na kontenerze Docker, by następnie przekierować połączenie na lokalny (względem serwera SSH, czyli też na kontenerze) serwer FTP. Program sygnalizuje, że tunel został wystawiony na porcie . Na prawym terminalu, połączenie się z tym portem lokalnym przekierowuje nas do serwera FTP. Sprawdźmy jeszcze w Wiresharku, jak wyglądają przesyłane pakiety.

Na Rysunku 5.8 znajdują się przechwycone w Wiresharku pakiety. Jak widać, komunikacja jest zaszyfrowana. Możesz dla porównania podsłuchać pakiety, gdy łączysz się z dowolnym serwerem SSH.

Dla pewności, na Rysunku 5.9 umieściłem przechwycone pakiety w sytuacji, gdy łączymy się bezpośrednio z serwerem FTP, bez użycia tunelu. Jak widać, w treści pakietów można podejrzeć login i hasło wykorzystane do połączenia się z serwerem FTP.

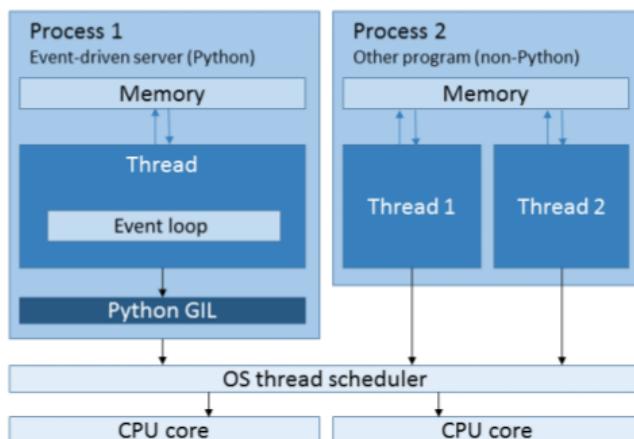
Tunel SSH bywa bardzo przydatny, więc warto wspomnieć, że można go zestreścić w prosty sposób bez wykorzystania skryptu, którego użyłem do omówienia pojęcia. Wystarczy uruchomić komendę `ssh uzytkownik@adres-serwera -D 8080`, która połączy się z serwerem SSH i na lokalnym porcie 8080 uruchomi usługę pośrednika SOCKS5. Wystarczy w przeglądarce ustawić adres proxy na `localhost:8080` i cała komunikacja będzie szyfrowana i będzie przechodziła przez serwer SSH. Polecam również flagi `-L` i `-R`.

5.3 Serwery zdarzeniowe

Wielowątkowość ma wiele zalet, w szczególności łatwość programowania i obsługi, ponieważ programy wielowątkowe są zbliżone do jednowątkowych poza fragmentami, gdzie należy zsynchronizować dostęp do współdzielonych danych. Jednakże dużym minusem jest potrzeba utrzymania wątku w przypadku serwerów, które obsługują dużą liczbę klientów. Do każdego wątku przydzielona jest pamięć podrzeczna, a przełączanie procesora między wątkami (ang. context switching) dla ich dużej liczby staje stosunkowo kosztowne.

Alternatywą dla wielowątkowości jest wykorzystanie modelu zdarzeniowego (ang. event-driven). W modelu tym to nie system operacyjny automatycznie i samowolnie przełącza się między aktywnymi wątkami lub procesami. Program wykorzystuje jeden wątek, który rejestruje w systemie operacyjnym obiekty blokowane (np. gniazda). Kiedy obiekty te zmieniają stan z blokowanego na aktywny (np. kiedy gniazdo otrzyma dane) system operacyjny powiadamia program (występuje zdarzenie), który może wykonywać operację na obiekcie bez potrzeby synchronizacji, ponieważ zdarzenie jest gwarantem, że obiekt opuścił stan blokowany. Ponadto wszelkie operacje na obiekcie są wykonywane natychmiastowo (nie blokują działania), ponieważ obiekt nie jest w stanie zablokowanym. Program wykorzystuje pętlę, w której sprawdza, czy system operacyjny powiadomił, że jakiś obiekt został odblokowany. Wtedy obsługujemy takie zdarzenie i wracamy do pętli. Pętlę tą nazywamy pętlą zdarzeń (ang. event loop).

Takie podejście ma podobną wydajność do rozwiązania wielowątkowego, jednakże nie wymaga przydziału pamięci podrzecznej oraz przełączania wątków, dzięki czemu jest bardziej skalowane.



Rysunek 5.10: Obsługa zdarzeń - architektura. Źródło: [7].

Na Rysunku 5.10 umieszczono porównanie między programami w języku Python, wykorzystującymi pętlę zdarzeń oraz programami wielowątkowymi nie napisanymi w języku Python. W tym przypadku GIL nie ma wpływu na wy-

dajność, ponieważ program wykorzystuje tylko jeden wątek.

5.3.1 Chat zdarzeniowy

Wcześniej wspomnieliśmy, że programy zdarzeniowe mają wiele zalet. Problem polega na tym, że programowanie w modelu zdarzeniowym jest całkiem inne, niż w przypadku wielowątkowości. Należy zrezygnować z niskiego poziomu implementacji. W tym podrozdziale napiszemy chat oparty na zdarzeniach.

W przypadku chatu zdarzeniowego wykorzystamy interfejs `poll`, który istnieje tylko w Linuxowej wersji Pythona. Istnieje starsza implementacja obsługi zdarzeń, `select`, jednakże jest ona wolniejsza i bardziej skomplikowana. Frameworki, o których powiemy sobie później, automatycznie wykrywają system operacyjny i przełączają się na interfejs `select` w przypadku systemu Windows.

Obok `poll` istnieje interfejs `epoll`, również dostępny na systemie Linux, który jest bardziej wydajny. Jednakże jest on również bardziej skomplikowany w użyciu, dlatego dla uproszczenia pozostaniemy przy interfejsie `poll`.

Uwaga! Interfejs `poll`, z którego będziemy korzystać znajduje się w module `select`, dlatego to ten moduł będziemy importować. Należy jednak pamiętać, że to nie oznacza, że korzystamy z interfejsu `select`.

W tym podrozdziale ponownie skorzystamy z naszej aplikacji chat, tylko przebudujemy ją na wersję opartą na zdarzeniach. Analogicznie do szyfrowanego chatu tutaj również omówimy jedynie różnice, jednakże będzie ich trochę więcej, ponieważ podejście zdarzeniowe jest całkowicie inne od wielowątkowego.

Na początek prześledźmy zmiany w kodzie serwera.

```

1 import select
2 from types import SimpleNamespace
3 from collections import deque
4
5 (...)

7 clients = {}

9 if __name__ == '__main__':
10     listen_sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
11     listen_sock.bind(("0.0.0.0", port))
12     listen_sock.listen(5)
13     poll = select.poll()
14     poll.register(listen_sock, select.POLLIN)

```

Listing 5.13: Główny program serwera - obiekt `poll`. Źródło: [7].

Podstawowa różnica względem serwera wielowątkowego jest taka, że w tym przypadku nie mamy synchronizacji, ponieważ program zawiera tylko jeden wątek. W związku z tym nie ma żadnych obiektów z modułu `threading`, takich

jak Lock. Tworzony jest natomiast jeden obiekt poll z modułu select, który będzie zbiorem zdarzeń. Obiekt ten zbiera zdarzenia I/O (input/output) powiązane z deskryptorami plików (np. gniazda), które oznaczają zmianę ich stanu. Na przykład zdarzeniem może być informacja, że dane gniazdo jest gotowe do odebrania lub wysłania danych. Dostępne są następujące zdarzenia I/O:

- POLLIN - deskryptor gotowy od odbioru danych.
- POLLOUT - deskryptor gotowy od wysłania danych w trybie nieblokującym.
- POLLPRI - deskryptor gotowy do odbioru danych o podwyższonym priorytecie.
- POLLERR - zdarzenie oznaczające błąd deskryptora.
- POLLHUP - zdarzenie zerwania połączenia powiązanego z deskryptorem.
- POLLNVAL - zdarzenie oznacza zamknięty deskryptor.

Wywołanie metody register na obiekcie poll powoduje, że zbiera on zdarzenia typu select.POLLIN na gnieździe nasłuchującym na nowych klientów.

```

1 while True:
2
3     # Iterate over all sockets with events
4     for fd, event in poll.poll():
5         # clear-up a closed socket
6         if event & (select.POLLHUP | select.POLLERR | select.POLLNVAL):
7             poll.unregister(fd)
8             del clients[fd]
9
10    # Accept new connection, add client to clients dict
11    elif fd == listen_sock.fileno():
12        client_sock,addr = listen_sock.accept()
13        client_sock.setblocking(False)
14        fd = client_sock.fileno()
15        clients[fd] = create_client(client_sock)
16        poll.register(fd, select.POLLIN)
17        print('Connection from {}'.format(addr))
18
19    (...)
```

Listing 5.14: Główny program serwera - pętla główna. Źródło: [7].

W dalszej kolejności serwer uruchamia pętle nieskończoną, w której będzie obsługiwał zdarzenia. Obsługa polega na pobraniu wszystkich zdarzeń z obiektu poll za pomocą metody poll(). Każdy wynik składa się z deskryptora pliku fd oraz typu zdarzenia event.

Na początku skrypt sprawdza, czy nie zostało zakończone połączenie z klientem, co jest realizowane za pomocą porównania maskowanego (operator &) z wartościami POLLHUP, POLLERR oraz POLLNVAL. W takiej sytuacji z obiektu poll usuwany jest deskryptor tego połączenia (gniazda) oraz jest on usuwany ze zmiennej clients, przechowującej informację o połączonych klientach.

Drugi sprawdzany warunek to taki, czy deskryptor zdarzenia nie pochodzi z gniazda nasługiującego na klientów. Jeśli tak, to oznacza to, że połączył się nowy klient. Wtedy akceptowane jest połączenie za pomocą metody accept w wyniku czego pojawia się nowe gniazdo client_sock połączone z klientem. Gniazdo oznaczane jest jako nieblokujące (ang. non-blocking) za pomocą metody setblocking. Gniazdo nie blokujące to takie, które w przypadku wywołania metody recv nie oczekuje na dane, tylko sprawdza czy są one dostępne, a jeśli danych w tej chwili nie ma, to kończy działanie.

```

1 def create_client(sock):
2     """ Return an object representing a client """
3     return SimpleNamespace(sock=sock, rest=bytes(),
4                           send_queue=deque(), name=None)

```

Listing 5.15: Funkcja create_client. Źródło: [7].

Po podłączeniu się nowego klienta tworzony jest nowy obiekt z jego gniazdem sock, nierożkodowanym fragmentem danych rest, kolejką danych do wysłania send_queue oraz nazwy name. Obiekt ten zapisywany jest w globalnej zmiennej clients pod indeksem deskryptora gniazda klienta. Dostęp do globalnej zmiennej nie jest synchronizowany, bo przecież ciągle jesteśmy w jednym wątku.

Na koniec, do obiektu poll dodawana jest informacja (metoda register), aby nasłuchiwał on na zdarzenia związane z gniazdem klienta, które oznaczają, że są gotowe dane do odebrania (POLLIN).

```

1     (..)
2     # Handle received data on socket
3     elif event & select.POLLIN:
4         client = clients[fd]
5         addr = client.sock.getpeername()
6         try:
7             msgs, client.rest = recv_msg(client.sock, client.rest)
8         except ConnectionError:
9             # the client state will get cleaned up in the
10            # next iteration of the event loop, as close()
11            # sets the socket to POLLNVAL
12            client.sock.close()
13            print('Client {} disconnected'.format(addr))
14            continue
15        # If we have any messages, broadcast them to all
16        # clients
17        for msg in msgs:
18            print('{}: {}'.format(addr, msg))
19
20        # Handle first message. Set clients name.
21        if client.name is None:
22            client.name = msg
23        else:
24            """ Add message to each connected client's send queue """
25            msg = '{}: {}'.format(client.name, msg)
26            for i in clients:
27                clients[i].send_queue.append(prep_msg(msg))
28                poll.register(clients[i].sock, select.POLLOUT)

```

Listing 5.16: Główny program serwera - pętla główna (cd.). Źródło: [7].

Kolejnym rodzajem zdarzenie, które jest obsługiwane w pętli głównej to zdarzenie POLLIN, czyli gotowość (jakiegoś) gniazda do odebrania danych. W tej

sytuacji pobieramy obiekt klienta i z jego gniazda wczytujemy i dekodujemy wiadomości za pomocą funkcji `recv_msg`. Nierożkodowany fragment zapisujemy w polu `rest` klienta. Jeśli w momencie pobierania danych wystąpił wyjątek, to zamykamy połączenie, a dane o kliencie zostaną usunięte w ramach obsługi kolejnego zdarzenia.

Po zdekodowaniu wiadomości `msgs` każda z nich jest wyświetlana na ekranie (logowanie po stronie serwera). Następnie serwer sprawdza, czy jest to pierwsza wiadomość od klienta i jeśli tak, to zapisuje jej treść jako nazwę klienta. W przeciwnym razie wiadomość jest kodowana do takiej formy, w jakiej będzie przesyłana i dodawana jest do kolejki wiadomości do wysłania każdego klienta. Na końcu obiekt `poll` otrzymuje polecenie, aby nasłuchiwał na zdarzenia mówiące o tym, że gniazdo klienta jest gotowe do wysłania danych.

```

1 (...) 
2 # Send message to ready client
3 elif event & select.POLLOUT:
4     client = clients[fd]
5     data = client.send_queue.popleft()
6     sent = client.sock.send(data)
7     if sent < len(data):
8         client.rest.appendleft(data[sent:])
9     if not client.send_queue:
10        poll.modify(client.sock, select.POLLIN)

```

Listing 5.17: Główny program serwera - pętla główna (cd.). Źródło: [7].

Ostatni `if` w głównej pętli serwera sprawdza, czy wystąpiło zdarzenie, o którym wspomniałem w poprzednim akapicie, czyli zdarzenie gotowości gniazda klienta do wysłania (`POLLOUT`). Wtedy na podstawie deskryptora zdarzenia pobierany jest obiekt klienta ze zmiennej `clients` oraz najstarsza wiadomość do wysłania `data`. Następnie dane są wysyłane gniazdem klienta `client.sock` i sprawdzana jest liczba wysłanych bajtów. Jeśli jest ona mniejsza niż rozmiar oryginalnej wiadomości (gniazdo wysłało tylko fragment) to pozostała, niewysłana część wraca jako najstarsza wiadomość do kolejki wiadomości do wysłania `client.send_queue`. Natomiast jeśli kolejka do wysłania jest pusta (np. gdy w kolejnych obrotach pętli `while` ta gałąź wyśle wszystkie wiadomości z kolejki) to typ nasłuchiwanego zdarzeń z obiektu `poll` jest zamieniany dla tego klienta na `POLLIN`, czyli oczekiwanie na dane.

A co z klientem?

Klient chat pozostaje bez zmian, ponieważ klient wykorzystuje stale tylko 2 wątki. W przypadku serwera przejście na zdarzeniowość ma sens, ponieważ z każdym nowym klientem przybywają 2 wątki.

The styl programowania (wykorzystujący `poll` oraz nieblokujące gniazda) jest często opisywany jako nieblokujący i asynchroniczny, ponieważ program odpowiada na występujące zdarzenia wejścia-wyjścia, a nie oczekuje na konkretnym kanale wejścia do momentu, aż pojawią się dane. Opisany powyżej program nie jest jednak w 100% nieblokujący, ponieważ występuje w nim metoda `poll.poll()`, która czeka, aż pojawią się jakieś zdarzenia. Jest to jednak nieuniknione, ponieważ jeśli nikt nie rozmawia to żadne zdarzenie się nie pojawi, a program działa.

5.3.2 Chat zdarzeniony w asyncio

Asyncio to moduł wprowadzony w wersji 3.4 Pythona i jego zadanie było ustanowianie asynchronicznej obsługi wejścia-wyjścia. Program składa się ze współprogramów (ang. co-routine), którymi zarządza pętla główna. Główny program może do niej dodawać współprogramy, które są wykonywane asynchronicznie. Pętla zajmuje się szeregowaniem współprogramów oraz optymalizacją ich wykonania.

Moduł `asyncio` posiada wbudowaną obsługę gniazd, dzięki czemu stworzenie serwera staje się dosyć prostym zadaniem. Poniżej omówimy kod programu serwera, który wykorzystuje `asyncio`. Ponownie, kod klienta się nie zmienia względem rozwiązania wielowątkowego, ponieważ nie ma to sensu pod względem wydajności.

```

1 import sys
2 import socket
3 import asyncio
4
5 clients = []
6
7 (...)

8
9 class ChatServerProtocol(asyncio.Protocol):
10     """ Each instance of class represents a client and the socket
11     connection to it. """
12
13     def connection_made(self, transport):
14         """ Called on instantiation, when new client connects """
15         self.transport = transport
16         self.addr = transport.get_extra_info('peername')
17         self._rest = b''
18         self.name = None
19         clients.append(self)
20         print('Connection from {}'.format(self.addr))
21
22     def data_received(self, data):
23         """ Handle data as it's received. Broadcast complete
24         messages to all other clients """
25         data = self._rest + data
26         (msgs, rest) = parse_recd_data(data)
27         self._rest = rest
28         for msg in msgs:
29             msg = msg.decode('utf-8')
30             if self.name is None:
31                 self.name = msg
32             else:
33                 msg = '{}: {}'.format(self.name, msg)
34             print(msg)
35             msg = prep_msg(msg)
36             for client in clients:
37                 client.transport.write(msg) # <-- non-blocking
38
39     def connection_lost(self, ex):
40         """ Called on client disconnect. Clean up client state """
41         print('Client {} disconnected'.format(self.addr))
42         clients.remove(self)
43
44 if __name__ == '__main__':
45     if len(sys.argv) != 2:
46         sys.stderr.write("usage: chat_server port\n")
47         exit(1)
48
49 try:

```

```

50     port = int(sys.argv[1])
51     assert port > 0
52 except:
53     sys.stderr.write("error: invalid port\n")
54     exit(1)
55
56 loop = asyncio.get_event_loop()
57 # Create server and initialize on the event loop
58 coroutine = loop.create_server(ChatServerProtocol, host='0.0.0.0', port=port)
59 server = loop.run_until_complete(coroutine)
60 # print listening socket info
61 for socket in server.sockets:
62     addr = socket.getsockname()
63     print('Listening on {}'.format(addr))
64 # Run the loop to process client connections
65 loop.run_forever()

```

Listing 5.18: Główny program serwera. Źródło: [7].

Prześledźmy po kolej kod serwera. Na początku definiujemy klasę `ChatProtocol`, która odpowiada za działanie serwera. Dziedziczy ona po klasie `asyncio.Protocol` i definiuje protokół, czyli sposób zachowania serwera na zdarzenia takie jak nowe połaczenie, odebrane dane, czy zakończenie połączenia.

Klasa `Protocol` wymaga nadpisania metod `connection_made()`, `data_received()` oraz `connection_lost()`. Klasa ta zostanie dodana do głównej pętli zdarzeń i za każdym razem, gdy pojawi się nowe połaczenie, obiekt tej klasy będzie stworzony do obsługi tego połączenia. Klasa dodawana jest w programie głównym za pomocą metody `create_server` na obiekcie `poll`, zwróconym przez funkcję `get_event_loop` z modułu `asyncio`.

Metoda `connection_made()` jest wywoływana, gdy zostanie nawiązane nowe połaczenie. Jest to odpowiednik zdarzenia `POLLIN` z modułu `select` oraz metody `socket.accept()` z modułu gniazd. Parametr `transport` to abstrakcyjny obiekt pochodzący z modułu `asyncio` i reprezentujący strumień danych (`asyncio.WriteTransport`). Serwer będzie z niego korzystał do przesyłania danych przez gniazdo, więc zapisujemy go w polu `transport`. Skrypt zapisuje również adres oraz port klienta za pomocą metody `transport.get_extra_info('peername')` oraz ustala początkowe wartości dla nazwy klienta (`name`) oraz nierożkodowanych danych (`rest`) przez funkcję `parse_recv_data`. Na koniec klient (obiekt `ChatProtocol`) jest dodawany do globalnej listy klientów `clients`.

Metoda `data_received` jest wywoływana, gdy serwer otrzyma jakieś dane od klienta. Rozkodowuje ona wiadomości za pomocą funkcji `parse_recv_data`, które następnie są przesyłane do wszystkich klientów za pomocą metody `write` (nieblokującej) wywoływanej na ich polu `transport`.

Ostatnia metoda to `connection_lost()`, która jest wywoywana, gdy klient się rozłączy lub połaczenie zostanie zerwane (analogiczne do zwrócenia przez `recv` pustego napisu lub wystąpienia wyjątku `ConnectionError`). Jedynie co robi ta metoda, to usunięcie obiektu klienta z listy klientów.

W programie głównym tworzymy pętlę zdarzeń, po czym dodajemy do niej serwer wykorzystujący klasę `ChatProtocol` jako protokół. Następnie za pomocą metody `run_until_complete` oczekujemy na zakończenie uruchamiania serwera, aby pobrać jego obiekt i wyświetlić informację o adresie, na którym

nasłuchuje. Na końcu uruchamiamy nieskończoną pętlę zdarzeń za pomocą metody `run_forever`.

Bibliografia

- [1] Matt Walker, *CEH Certified Ethical Hacker All-in-One Exam Guide*, McGraw Hill, 2nd edition, 2014.
- [2] https://pl.wikipedia.org/wiki/Address_Resolution_Protocol, sierpień 2016.
- [3] https://pl.wikipedia.org/wiki/IP_Protocol, sierpień 2016.
- [4] https://pl.wikipedia.org/wiki/User_Datagram_Protocol, sierpień 2016.
- [5] https://pl.wikipedia.org/wiki/Transmission_Control_Protocol, sierpień 2016.
- [6] https://pl.wikipedia.org/wiki/Domain_Name_System, sierpień 2016.
- [7] Dr. M. O. Faruque Sarker, Sam Washington, *Learning Python Network Programming*, PACKT, 2014.
- [8] http://www.asawicki.info/Mirror/Beej_s%20Guide%20to%20Network%20Programming%20PL/bgnet.pdf, sierpień 2016.
- [9] <http://beej.us/guide/bgnet/output/html/multipage/htonsman.html>, sierpień 2016.
- [10] <https://sekurak.pl/bezpieczenstwo-protokolu-websocket-w-praktyce/>, marzec 2017.