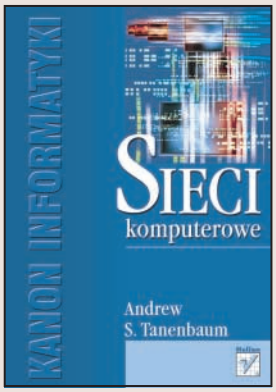
Sieci komputerowe



Autor: Andrew S. Tanenbaum
Tłumaczenie: Andrzej Grażyński, Adam Jarczyk
ISBN: 83-7361-557-1
Tytuł oryginału: Computer Networks, 4th Edition
Format: B5, stron: 804

Sieci komputerowe rozwijają się niezwykle dynamicznie. Regularnie pojawiają się nowe technologie, nowe sposoby przekazywania danych, nowe protokoły i narzędzia. Chyba każdy użytkownik komputera spotkał się przynajmniej raz z siecią komputerową. Internet jest dziś tak powszechny jak telewizja czy radio. Coraz popularniejsze stają się też nowe technologie — sieci bezprzewodowe, Bluetooth i sieci komórkowe. Znajomość zagadnień leżących u podstaw projektowania i wykorzystywania sieci komputerowych jest przydatna każdemu, kto chce być na bieżąco z nowoczesnymi technologiami.

„Sieci komputerowe" to kompendium wiedzy poświęcone współczesnym technologiom sieciowym. Opisuje zarówno mechanizmy już wykorzystywane, jak i te, które są obecnie w fazie badań i testów. Przedstawia sieci kablowe i bezprzewodowe oraz wykorzystujące je aplikacje — WWW, radio internetowe, usługi sieciowe i wiele innych.

- Zastosowania sieci komputerowych
- Warstwa fizyczna — kable, światłowody i łącza bezprzewodowe
- Warstwa łącza danych — protokoły sieciowe, weryfikacja poprawności przesyłu danych
- Kontrola dostępu do nośnika
- Podwarstwa MAC – Gigabit Ethernet, 802.11, szerokopasmowy dostęp bezprzewodowy i przełączanie
- Warstwa sieciowa — algorytmy routingu, kontrola przeciążeń, QoS, IPv4 i IPv6
- Warstwa transportowa — programowanie gniazd, UDP, TCP, RTP i wydajność sieci
- Warstwa aplikacji — e-mail, WWW, PHP, bezprzewodowy dostęp do WWW, MP3 i strumieniowe przesyłanie dźwięku
- Bezpieczeństwo sieci — AES, RSA, kryptografia kwantowa, IPsec i bezpieczeństwo WWW

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

Spis treści

3.

Warstwa łącza danych....167

4.

Podwarstwa kontroli dostępu do nośnika....221

8.

Bezpieczeństwo w sieciach komputerowych 641

9.

3
Warstwa łącza danych

W niniejszym rozdziale poznamy zasady projektowania warstwy 2. — warstwy łącza danych. Zajmiemy się algorytmami zapewniającymi niezawodną, skuteczną komunikację pomiędzy dwoma sąsiadującymi komputerami na poziomie tej warstwy. Przez „sąsiadujące" rozumiemy dwa komputery połączone kanałem komunikacyjnym, który zachowuje się jak przewód (np. kabel koncentryczny, linia telefoniczna lub dwupunktowy kanał bezprzewodowy). Istotną właściwością, która powoduje, że kanał zachowuje się jak „kabel", jest to, że bity są doręczane w dokładnie tej samej kolejności, w której były wysłane.

Na pierwszy rzut oka ten problem może się wydawać tak trywialny, że nie trzeba żadnego oprogramowania, z którym musielibyśmy się zapoznać — komputer *A* po prostu wysyła bity kablem, a komputer *B* je odbiera. Niestety, w kanałach komunikacyjnych występują od czasu do czasu błędy. Co więcej, każdy kanał ma skończoną przepływność i niezerowe opóźnienie propagacji pomiędzy chwilą wysłania a chwilą odbioru bitu. Te ograniczenia mają znaczący wpływ na wydajność przesyłu danych. Protokoły używane do komunikacji muszą uwzględniać wszystkie te czynniki. Właśnie te protokoły są tematem niniejszego rozdziału.

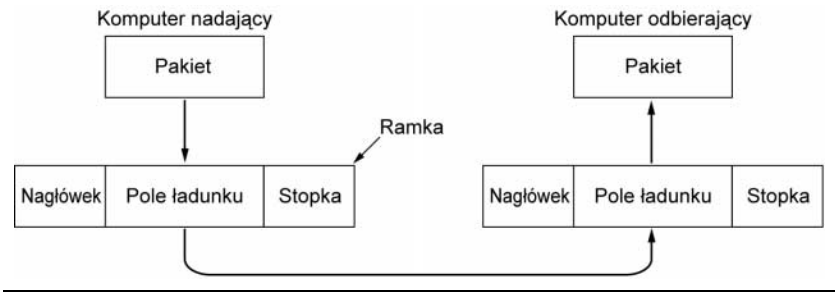
Po wprowadzeniu do podstawowych problemów projektowych warstwy łącza danych zaczniemy analizę jej protokołów od poznania charakteru błędów, ich przyczyn i sposobów, w jakie mogą być wykrywane i korygowane. Następnie przestudiujemy kilka protokołów o rosnącej złożoności, z których każdy rozwiązuje coraz więcej problemów występujących w tej warstwie. Zakończymy opisem modelowania i poprawności protokołów oraz podamy kilka przykładów protokołów łącza danych.

3.1. Problemy projektowe warstwy łącza danych

Warstwa łącza danych może spełniać kilka określonych funkcji, w tym:

(1) Udostępniać dobrze zdefiniowany interfejs usługi warstwie sieciowej.
(2) Radzić sobie z błędami transmisji.
(3) Sterować przepływem danych, aby wolne odbiorniki nie zostały zalane danymi z szybkich nadajników.

Aby zrealizować te cele, warstwa łącza danych bierze pakiety, które otrzymuje z warstwy sieciowej, i kapsułkuje je do transmisji w **ramki**. Każda ramka zawiera nagłówek, pole treści zasadniczej i stopkę, jak na rysunku 3.1. Zarządzanie ramkami jest głównym zadaniem warstwy łącza danych. W następnych punktach zapoznamy się szczegółowo z problemami wymienionymi powyżej.



RYSUNEK 3.1. Relacje pomiędzy pakietami i ramkami

Wprawdzie niniejszy rozdział jest poświęcony warstwie łącza danych i protokołom łącza danych, lecz wiele zasad, które tu poznamy, takich jak kontrola błędów i sterowanie przepływem, spotkamy również w protokołach transportowych i innych. W rzeczy samej w wielu sieciach funkcje te są obecne tylko w wyższych warstwach, a nie w warstwie łącza danych. Jednakże niezależnie od ich położenia zasady są praktycznie takie same, więc tak naprawdę nie jest istotne, kiedy się nimi zajmiemy. W warstwie łącza danych często pojawiają się one w najprostszych i najczystszych formach, dlatego też jest to dobre miejsce, by przyjrzeć im się dokładnie.

3.1.1. Usługi świadczone dla warstwy sieciowej

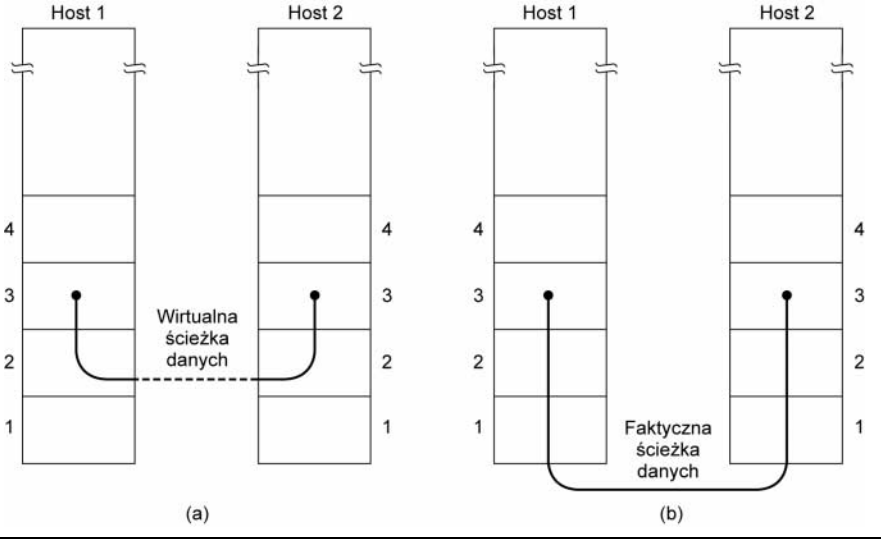
Funkcją warstwy łącza danych jest świadczenie usług warstwie sieciowej. Podstawowa usługa to transfer danych z warstwy sieciowej w komputerze źródłowym do warstwy sieciowej w komputerze docelowym. W komputerze źródłowym w warstwie sieciowej znajduje się coś, nazwijmy to procesem, co przekazuje do warstwy łącza danych bity przeznaczone do przesłania do celu. Zadaniem warstwy łącza danych jest przesłanie bitów do komputera docelowego, aby mogły zostać tam przekazane do warstwy sieciowej, jak na rysunku 3.2 (a). Faktyczna transmisja odbywa się ścieżką z rysunku 3.2 (b), lecz łatwiej jest myśleć kategoriami dwóch procesów warstwy łącza danych komunikujących się za pomocą protokołu łącza danych. Z tego powodu będziemy w całym tym rozdziale domyślnie używać modelu z rysunku 3.2 (a).

Warstwa łącza danych może być zaprojektowana tak, że będzie oferować różne usługi. Faktyczna lista usług może być różna zależnie od systemu. Trzy opcje, które są powszechnie udostępniane, to:

(1) Usługa bezpołączeniowa bez potwierdzeń.
(2) Usługa bezpołączeniowa z potwierdzeniami.
(3) Usługa połączeniowa z potwierdzeniami.

Spójrzmy kolejno na każdą z nich.

W usłudze bezpołączeniowej bez potwierdzeń komputer źródłowy wysyła niezależne ramki do komputera docelowego bez potwierdzania ich przez komputer docelowy. Żadne logiczne połączenie nie jest nawiązywane przed transmisją i zwalniane po transmisji. Jeśli ramka zostanie utracona z powodu zakłóceń na linii, w warstwie łącza danych nie jest podejmowana żadna próba wykrycia straty ani przywrócenia ramki. Ta klasa usług nadaje się tam, gdzie stopa błędów jest bardzo niska, więc naprawę pozostawia się warstwom wyższym. Jest też stosowana w transmisji w czasie rzeczywistym, na przykład mowy, gdzie opóźnienia danych są gorsze od błędów. Większość sieci LAN używa takiej usługi w warstwie łącza danych.



RYSUNEK 3.2. (a) Komunikacja wirtualna. (b) Komunikacja faktyczna

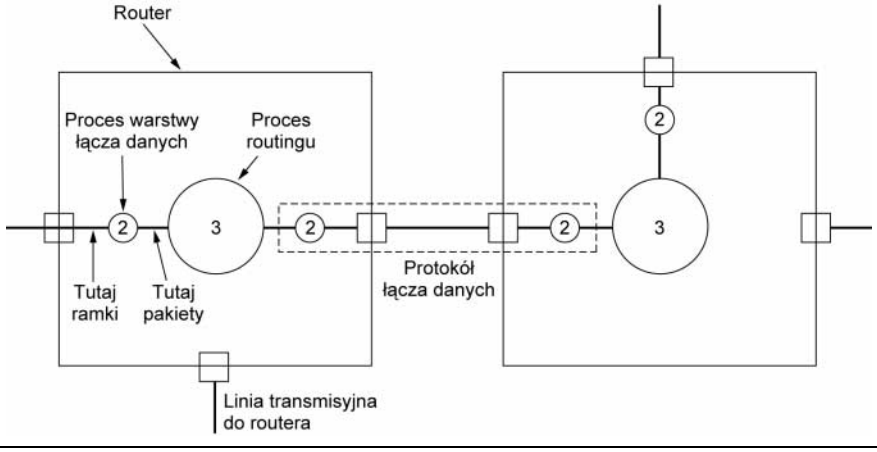
Następnym poziomem (w sensie niezawodności) jest usługa bezpołączeniowa z potwierdzeniami. Gdy usługa ta jest oferowana, nadal nie stosuje się logicznych połączeń, lecz każda wysłana ramka jest indywidualnie potwierdzania. Dzięki temu nadawca wie, czy ramka dotarła bez błędów. Jeśli ramka nie dotrze w ustalonym czasie, można ją nadać ponownie. Ta usługa jest przydatna dla zawodnych kanałów, takich jak systemy bezprzewodowe.

Warto może podkreślić, że udostępnienie potwierdzeń w warstwie łącza danych jest jedynie optymalizacją, a nigdy wymogiem. Warstwa sieciowa zawsze może wysłać pakiet i czekać na jego potwierdzenie. Jeśli potwierdzenie nie wróci przed upływem określonego czasu, nadajnik może wysłać jeszcze raz cały komunikat. Problem z tą strategią polega na fakcie, że ramki mają zwykle ściśle określoną maksymalną długość narzuconą przez sprzęt, a pakiety warstwy sieciowej nie. Jeśli przeciętny pakiet podzielimy np. na 10 ramek, a 20% wszystkich ramek będzie traconych, to może potrwać bardzo długo, zanim cały pakiet zostanie przesłany. Gdy potwierdzane i nadawane ponownie są pojedyncze ramki, całe pakiety przechodzą znacznie szybciej. W niezawodnych kanałach, takich jak światłowodowe, dodatkowe obciążenie wprowadzane przez „opasły" protokół łącza danych może być niepotrzebne, lecz w kanałach bezprzewodowych, które z natury są zawodne, jest to warte swojej ceny.

Wracając do usług, najbardziej zaawansowaną, jaką może udostępnić warstwa łącza danych warstwie sieciowej, jest usługa połączeniowa. W takiej usłudze komputery źródłowy i docelowy nawiązują połączenie przed przesłaniem jakichkolwiek danych. Każda ramka wysłana takim połączeniem ma swój numer, a warstwa łącza danych gwarantuje, że każda wysłana ramka zostanie faktycznie odebrana. Co więcej, gwarantuje, że każda ramka zostanie odebrana raz i tylko raz oraz że wszystkie ramki zostaną odebrane we właściwej kolejności. W przeciwieństwie do tego w usłudze bezpołączeniowej może się zdarzyć, że utracone potwierdzenie spowoduje wysłanie i odebranie jednego pakietu kilka razy. Usługa połączeniowa daje procesom warstwy sieciowej odpowiednik niezawodnego strumienia bitów.

Przy użyciu usługi połączeniowej transfer ma trzy wyraźne fazy. W pierwszej zostaje nawiązane połączenie przez inicjalizację przez obie strony zmiennych i liczników, potrzebnych do śledzenia na bieżąco, które ramki zostały odebrane, a które nie. W drugiej fazie zostaje faktycznie przesłana jedna lub więcej ramek. W trzeciej i ostatniej fazie połączenie zostaje zakończone, zwalniając zmienne, bufory i inne zasoby używane do utrzymywania połączenia.

Spójrzmy na typowy przykład: podsieć WAN składającą się z routerów połączonych dwupunktowymi liniami telefonicznymi. Gdy ramka dociera do routera, sprzęt sprawdza, czy nie zawiera ona błędów (z użyciem technik, które poznamy w dalszej części rozdziału), a następnie przekazuje ramkę do oprogramowania warstwy łącza danych (które może być wbudowane w układ scalony na karcie interfejsu sieciowego). Oprogramowanie warstwy łącza danych sprawdza, czy to jest oczekiwana ramka, a jeśli tak, przekazuje do oprogramowania rotingu pakiet zawarty w polu ładunku. Oprogramowanie rutingu wybiera następnie linię wyjściową i przekazuje pakiet z powrotem w dół do oprogramowania warstwy łącza danych, które wysyła pakiet. Rysunek 3.3 przedstawia przebieg danych przez dwa routery.



RYSUNEK 3.3. Miejsce protokołu łącza danych

Kod rutingu często wymaga, żeby transmisja została przeprowadzona poprawnie, czyli za pomocą niezawodnych, uporządkowanych połączeń na obu końcach łączy dwupunktowych. Nie lubi on, gdy po drodze gubi się zbyt wiele pakietów. Uczynienie z zawodnych linii komunikacyjnych łączy wyglądających na idealne (a przynajmniej przyzwoite) jest zadaniem protokołu łącza danych przedstawionego w prostokącie linią przerywaną. Tak na marginesie, chociaż pokazaliśmy w każdym routerze dwie kopie oprogramowania warstwy łącza danych, to naprawdę jedna kopia obsługuje wszystkie linie, przy czym dla każdej linii używa innych tablic i struktur danych.

3.1.2. Ramkowanie

Aby świadczyć usługi na rzecz warstwy sieciowej, warstwa łącza danych musi używać usług, które udostępnia jej warstwa fizyczna. Ta przyjmuje surowy strumień bitów i próbuje doręczyć go do miejsca przeznaczenia. Nie ma gwarancji, że ten strumień bitów będzie pozbawiony błędów. Liczba bitów odebranych może być mniejsza, równa lub większa od liczby bitów nadanych, a poszczególne bity mogą mieć różne wartości. Wykrywanie i w razie konieczności naprawa błędów należy do warstwy łączy danych.

Zazwyczaj robi się to tak, że warstwa łącza danych dzieli strumień bitów na dyskretne ramki i oblicza sumę kontrolną dla każdej ramki (algorytmy sum kontrolnych omówimy w dalszej części rozdziału). Gdy ramka dociera do miejsca przeznaczenia, suma kontrolna jest obliczana ponownie.

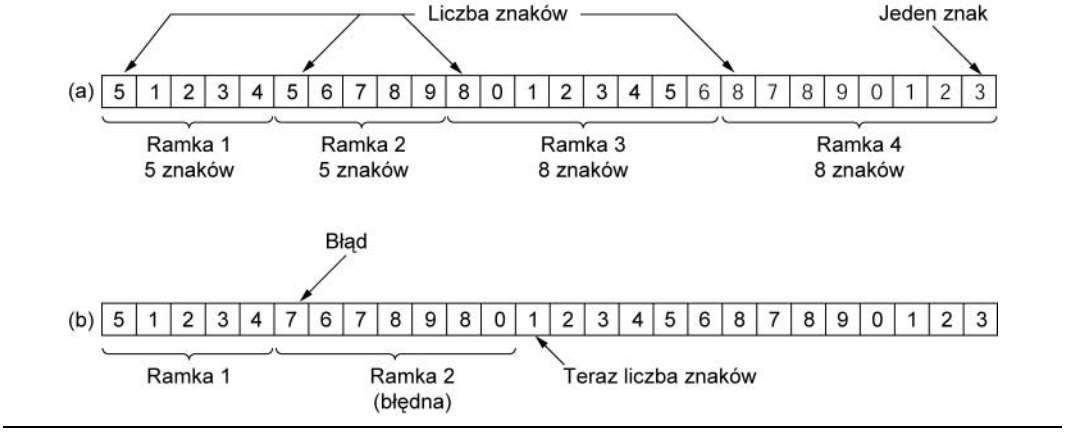
Jeśli wyliczona suma kontrolna różni się od tej zawartej w ramce, warstwa łącza danych rozpoznaje, że wystąpił błąd, i podejmuje odpowiednie kroki (np. odrzuca błędną ramkę i, ewentualnie, odsyła z powrotem informację o błędzie).

Podział strumienia bitów na ramki jest trudniejszy, niż się na pierwszy rzut oka wydaje. Jednym ze sposobów ramkowania może być wstawianie przerw czasowych pomiędzy ramki, podobnie jak wstawia się spacje pomiędzy słowa w zwykłym tekście. Jednakże sieci rzadko kiedy gwarantują utrzymanie zależności czasowych, więc może zdarzyć się, że te przerwy zostaną ściśnięte do zera lub podczas transmisji zostaną wprowadzone nowe przerwy.

Ponieważ liczenie na zależności czasowe do oznaczania początku każdej ramki jest ryzykowne, zostały opracowane inne metody. W niniejszym punkcie przyjrzymy się czterem metodom:

(1) Zliczanie znaków.
(2) Bajty znacznikowe z napełnianiem bajtami.
(3) Znaczniki początku i końca z napełnianiem bitami.
(4) Zmiany kodowania w warstwie fizycznej.

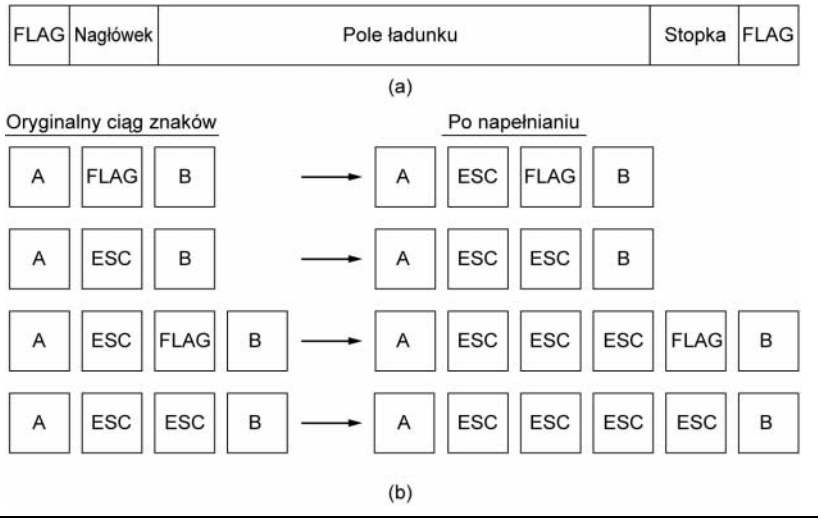
Pierwsza metoda ramkowania używa pola w nagłówku do oznaczenia liczby znaków w ramce. Gdy warstwa łącza danych w urządzeniu docelowym widzi zliczenie znaków, to wie, jak wiele znaków będzie następować w ramce, a więc również gdzie skończy się ramka. Rysunek 3.4 (a) przedstawia tę technikę dla czterech ramek o długości kolejno 5, 5, 8 i 8 znaków.



RYSUNEK 3.4. Strumień znaków: (a) bez błędów, (b) z jednym błędem

Problem z tym algorytmem polega na tym, że liczba może zostać przekłamana przez błąd transmisji. Na przykład, jeśli zliczenie znaków równe 5 w drugiej ramce z rysunku 3.4 (b) stanie się równe 7, to odbiornik utraci synchronizację i nie będzie mógł zlokalizować następnej ramki. Nawet jeśli suma kontrolna będzie niewłaściwa, przez co odbiornik wykryje błąd, nadal nie będzie mógł ustalić, gdzie zaczyna się następna ramka. Odesłanie ramki do źródła z żądaniem retransmisji też nie pomoże, ponieważ odbiornik nie wie, ile znaków musi pominąć, aby wrócić do początku retransmisji. Z tego powodu metoda ze zliczaniem znaków jest już używana bardzo rzadko.

Druga metoda rozwiązuje problem ponownej synchronizacji po błędzie przez rozpoczynanie i kończenie każdej ramki specjalnymi bajtami. W przeszłości bajty początku i końca były różne, lecz w ostatnich latach większość protokołów używa tego samego bajta, zwanego **bajtem znacznikowym** (ang. *flag byte*), jako separatora początkowego i końcowego oznaczonego na rysunku 3.5 (a)



RYSUNEK 3.5. (a) Ramka ograniczona bajtami znacznikowymi. (b) Cztery przykłady sekwencji bajtów przed i po napełnianiu bajtami

jako FLAG. W ten sposób, jeśli odbiorca kiedykolwiek utraci synchronizację, to wystarczy że poszuka bajta znacznikowego, aby znaleźć koniec bieżącej ramki. Dwa kolejne bajty znacznikowe wskazują koniec jednej ramki i początek następnej.

Poważny problem pojawia się w tej metodzie przy transmisji danych binarnych, na przykład programów wynikowych i liczb zmiennoprzecinkowych. Może się z łatwością zdarzyć, że wzorzec bajta znacznikowego wystąpi w danych. Ta sytuacja zwykle zakłóca ramkowanie. Jednym z rozwiązań problemu jest wstawianie przez warstwę łącza danych u nadawcy specjalnego bajta unikowego (ESC) przed każdy „przypadkowy" bajt znacznikowy w danych. Warstwa łącza danych po stronie odbierającej usuwa bajt unikowy przed przekazaniem danych do warstwy sieciowej. Ta technika nosi nazwę **napełniania bajtami** (**byte stuffing**) lub **napełniania znakami** (**character stuffing**). Dzięki niej bajt znacznikowy na końcu ramki może zostać odróżniony od takiego samego bajta danych przez obecność lub nieobecność bajta unikowego przed sobą.

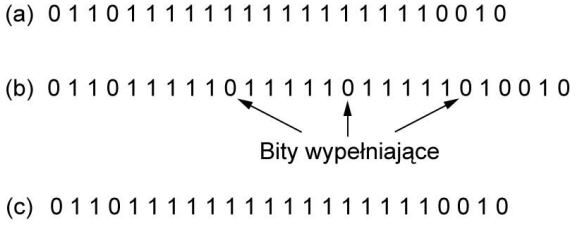
Oczywiście następne pytanie brzmi: co się stanie, gdy bajt unikowy wystąpi w środku danych? Odpowiedź: również zostanie uzupełniony o bajt unikowy. Wobec tego dowolny pojedynczy bajt unikowy jest elementem sekwencji unikowej, a dwa pod rząd oznaczają, że pojedynczy bajt unikowy wystąpił w danych w sposób naturalny. Kilka przykładów przedstawia rysunek 3.5 (b). W każdym przypadku sekwencja bajtów po usunięciu wypełnienia jest dokładnie taka sama jak oryginalna.

Schemat napełniania bajtami pokazany na rysunku 3.5 jest nieco uproszczoną wersją schematu używanego w protokole PPP, którego większość komputerów domowych używa do komunikacji ze swoim dostawcą usług internetowych. PPP omówimy w dalszej części rozdziału.

Poważną wadą tej metody ramkowania jest ścisły związek ze stosowaniem 8-bitowych znaków. Nie wszystkie metody kodowania używają 8-bitowych znaków. Na przykład w UNICODE znaki mają 16 bitów. W miarę rozwoju sieci wady stosowania długości kodu znaku w mechanizmach ramkowania zaczęły być coraz bardziej oczywiste, więc trzeba było opracować nową technikę — pozwalającą na używanie znaków o dowolnej długości.

Ta nowa technika pozwala umieszczać w ramce dowolną liczbę bitów i stosować kody znaków o dowolnej liczbie bitów na znak. Metoda działa tak: każda ramka zaczyna się i kończy specjalnym wzorcem bitów, 01111110 (w rzeczy samej to jest bajt znacznikowy). Zawsze, gdy warstwa łącza danych nadawcy napotyka w danych pięć jedynek pod rząd, automatycznie wstawia po nich „0" do wychodzącego strumienia bitów. To **napełnianie bitami** (ang. *bit stuffing*) przypomina napełnianie bajtami, w którym do wyjściowego strumienia znaków przed każdym bajtem znacznikowym występującym w danych był wstawiany bajt unikowy.

Gdy odbiornik otrzymuje pięć kolejnych bitów „1”, po których następuje „0”, automatycznie usuwa bit „0”. Napełnianie bitami jest całkowicie niewidoczne dla warstwy sieciowej w obu komputerach, podobnie jak napełnianie bajtami. Jeśli dane użytkownika zawierają wzorzec znacznikowy (01111110), to ten znacznik jest przesyłany jako 011111010, lecz w pamięci komputera odbierającego zostaje zapisany jako 01111110. Przykład napełniania bitami przedstawia rysunek 3.6.



RYSUNEK 3.6. Napełnianie bitami: (a) dane oryginalne, (b) postać tych danych w linii, (c) dane zapisane w pamięci odbiorcy po usunięciu wypełnienia

Napełnianie bitami pozwala rozpoznawać w sposób jednoznaczny granice pomiędzy ramkami za pomocą wzorca znacznika. Dzięki temu, gdy odbiornik utraci orientację, gdzie jest, wystarczy że poszuka w wejściowym strumieniu sekwencji znacznikowych, ponieważ mogą one występować tylko na granicach ramek, a nigdy w danych.

Ostatnia metoda ramkowania stosuje się tylko do sieci, w których kodowanie w nośniku fizycznym obejmuje pewną redundancję. Na przykład niektóre sieci LAN kodują 1 bit danych z użyciem 2 bitów fizycznych. Standardowo bit „1” jest reprezentowany przez parę wysoki-niski, a bit „0” przez parę niski-wysoki. Taki schemat oznacza, że każdy bit danych zawiera w środku przejście pomiędzy stanami ułatwiające odbiornikowi lokalizację granic bitów. Kombinacje wysoki-wysoki i niski-niski nie są używane w przesyłaniu danych, lecz w niektórych protokołach służą do rozdzielania ramek.

Na zakończenie tematu należy wspomnieć, że wiele protokołów łącza danych dla dodatkowego bezpieczeństwa używa kombinacji zliczania znaków z jedną z pozostałych metod. Gdy przychodzi ramka, pole zliczania służy do lokalizacji jej końca. Tylko wtedy, gdy na tej pozycji obecny jest odpowiedni znak rozdzielający i suma kontrolna jest poprawna, ramka zostaje zaakceptowana jako poprawna. W przeciwnym razie strumień wejściowy jest skanowany w poszukiwaniu kolejnego znaku rozdzielającego.

3.1.3. Kontrola błędów

Po rozwiązaniu problemu oznaczenia początku i końca każdej ramki przechodzimy do kolejnego problemu: jak upewnić się, że wszystkie ramki zostaną ostatecznie doręczone do warstwy sieciowej w miejscu przeznaczenia bez błędów i w odpowiedniej kolejności. Załóżmy, że nadajnik wysyła ramki jedna za drugą bez zwracania uwagi na to, czy docierają poprawnie do odbiornika. To rozwiązanie może być dobre dla usługi bezpołączeniowej bez potwierdzeń, lecz z pewnością nie nadaje się dla niezawodnych usług połączeniowych.

Niezawodne doręczanie zwykle zapewnia się przez zwracanie do nadawcy pewnych informacji o tym, co dzieje się na drugim końcu linii. Protokół zazwyczaj wymaga od odbiornika odesłania z powrotem specjalnych ramek sterujących potwierdzających pozytywnie lub negatywnie przychodzące ramki. Gdy nadajnik odbiera pozytywne potwierdzenie ramki, to wie, że ramka dotarła bezpiecznie. Z drugiej strony potwierdzenie negatywne oznacza, że coś poszło nie tak i że ramka musi zostać nadana jeszcze raz.

Dodatkowa komplikacja bierze się z możliwości, że problemy ze sprzętem mogą spowodować całkowite zniknięcie ramki (np. w impulsie zakłócającym). W takim przypadku odbiornik nie zareaguje w ogóle, ponieważ nie ma do tego żadnego powodu. Powinno być jasne, że protokół, w którym nadajnik wysyła ramkę i czeka na potwierdzenie, pozytywne lub negatywne, zawiesi się na dobre, gdy ramka zostanie utracona, na przykład z powodu wadliwego sprzętu.

Do radzenia sobie w takich sytuacjach służą czasomierze wprowadzone do warstwy łącza danych. Gdy nadajnik wysyła ramkę, zwykle również uruchamia czasomierz tak dobrany, że czas oczekiwania powinien wystarczyć na dotarcie ramki do celu, przetworzenie jej i propagację potwierdzenia z powrotem do nadajnika. Zazwyczaj ramka zostaje odebrana poprawnie i potwierdzenie wraca przed końcem zliczania czasu.

Jeśli jednak zgubi się ramka albo potwierdzenie, to czasomierz włącza się, informując nadajnik o potencjalnym problemie. Oczywistym rozwiązaniem jest wysłanie po prostu ramki jeszcze raz. Jednakże w przypadku wielokrotnego nadawania jednej ramki istnieje niebezpieczeństwo, że odbiornik przyjmie tę samą ramkę dwa lub więcej razy i przekaże ją do warstwy sieciowej więcej niż jeden raz. Aby temu zapobiec, trzeba zwykle przydzielić numery sekwencyjne wychodzącym ramkom, aby odbiornik mógł odróżnić ponowne transmisje od pierwotnych.

Cały problem zarządzania czasomierzami i numerami sekwencyjnymi tak, by zapewnić przekazanie w końcu u celu każdej ramki do warstwy sieciowej raz i tylko raz, jest ważną częścią obowiązków warstwy łącza danych. W dalszej części rozdziału przyjrzymy się kilku przykładom o coraz większej złożoności, aby zobaczyć, jak to wygląda w praktyce.

3.1.4. Sterowanie przepływem

Kolejnym ważnym problemem projektowym występującym w warstwie łącza danych (i wyższych warstwach też) jest sposób reagowania na sytuację, gdy nadajnik systematycznie chce wysyłać ramki szybciej, niż odbiornik będzie mógł je odbierać. Taka sytuacja może z łatwością wystąpić, gdy nadajnik pracuje w szybkim (lub mało obciążonym) komputerze, a odbiornik w komputerze wolnym lub przeciążonym. Nadajnik wysyła ramkę za ramką z dużą szybkością, dopóki odbiornik nie zostanie całkowicie zasypany danymi. Nawet jeśli transmisja jest wolna od błędów, w pewnym momencie odbiornik nie będzie po prostu w stanie przetwarzać dalszych nadchodzących ramek i zacznie je tracić. Coś trzeba oczywiście zrobić, aby zapobiec tej sytuacji.

Powszechnie stosowane są dwa podejścia. W **sterowaniu przepływem na podstawie informacji zwrotnych** nadajnik odsyła informacje do nadajnika, udzielając zgody na wysłanie dalszych danych lub przynajmniej informując nadajnik o stanie odbiornika. W drugiej metodzie, **sterowaniu przepływem opartym na szybkości transmisji**, protokół ma wbudowany mechanizm ograniczający szybkość, z jaką nadajniki mogą wysyłać dane, bez informacji zwrotnych z odbiornika. W tym rozdziale zajmiemy się schematami sterowania przepływem na podstawie informacji zwrotnych, ponieważ druga metoda nigdy nie jest stosowana w warstwie łącza danych. Schematom opartym na szybkości transmisji przyjrzymy się w rozdziale 5.

Znane są różne schematy sterowania przepływem na podstawie informacji zwrotnych, lecz większość z nich opiera się na tej samej podstawowej zasadzie. Protokół zawiera dobrze zdefiniowane reguły tego, kiedy nadajnik ma prawo wysłać kolejną ramkę. Reguły te często zabraniają wysyłania ramek bez zezwolenia ze strony nadajnika, wprost lub implikowanego. Na przykład przy nawiązywaniu połączenia odbiornik może zezwolić na wysłanie *n* ramek teraz, lecz kolejne dopiero po przysłaniu zezwolenia na kontynuację nadawania. Poniżej przyjrzymy się szczegółom tych schematów.

3.2. Wykrywanie i korekcja błędów

Jak widzieliśmy w rozdziale 2., system telefoniczny składa się z trzech części: central, łączy między centralami i pętli lokalnych. W większości krajów rozwiniętych dwa pierwsze składniki są niemal zawsze cyfrowe. Pętle lokalne nadal są analogowymi skrętkami miedzianymi i pozostaną w tej formie jeszcze przez lata z uwagi na ogromne koszty wymiany. Podczas gdy w części cyfrowej błędy są rzadkie, to nadal często występują w pętlach lokalnych. Co więcej, komunikacja bezprzewodowa staje się coraz popularniejsza, a w niej stopy błędów są o całe rzędy wielkości wyższe niż w łączach światłowodowych pomiędzy centralami. Wniosek: błędy transmisji były, są i będą z nami jeszcze przez wiele lat. Musimy nauczyć się, jak z nimi sobie radzić.

Z uwagi na właściwości procesów fizycznych, które je powodują, błędy w pewnych nośnikach (np. radiowych) pojawiają się seriami, a nie pojedynczo. Błędy tego typu mają zarówno wady, jak i zalety w porównaniu z odosobnionymi błędami jednobitowymi. Z jednej strony dane komputerowe są zawsze wysyłane w blokach bitów. Załóżmy, że blok ma wielkość 1000 bitów, a stopa błędów wynosi 0,001 na bit. Gdyby błędy były niezależnie, większość bloków zawierałaby błąd. Jednakże w przypadku błędów pojawiających się w seriach po 100 naraz, średnio tylko jeden lub dwa bloki na 100 zostałyby uszkodzone. Problem z seriami błędów polega na tym, że są znacznie trudniejsze do naprawy niż błędy odizolowane.

3.2.1. Kody korekcyjne

Projektanci sieci opracowali dwie podstawowe strategie radzenia sobie z błędami. Jednym sposobem jest zawarcie w każdym wysłanym bloku danych wystarczającej ilości redundantnych informacji, by odbiornik mógł odtworzyć oryginalne dane. Drugi sposób polega na zawarciu tylko takiej redundancji, która pozwala odbiornikowi wykryć wystąpienie błędu bez jego rozpoznania i zażądać ponownego wysłania danych. Pierwsza strategia wykorzystuje **kody korekcyjne** (ang. *error-correcting code*), a druga **kody detekcyjne** (**error-detecting code**). Korzystanie z kodów korekcyjnych często nosi nazwę **bezpośredniej korekcji błędów** (ang. *forward error correction*).

Każda z tych technik zajmuje inną niszę. W kanałach o wysokiej niezawodności, na przykład światłowodowych, taniej jest użyć kodu detekcyjnego i po prostu od czasu do czasu przesłać ponownie blok z wykrytym błędem. Jednakże w kanałach powodujących wiele błędów, takich jak np. łącza bezprzewodowe, lepiej dodać do każdego bloku wystarczającą redundancję, aby odbiornik mógł odtworzyć pierwotny blok, zamiast polegać na retransmisji, która też może zawierać błąd.

Aby zrozumieć, jak można radzić sobie z błędami, musimy przyjrzeć się dokładnie, czym w rzeczywistości jest błąd. Zwykle ramka składa się z m bitów danych (czyli komunikatu) i r bitów redundantnych (kontrolnych). Niech całkowita długość ramki wynosi n (czyli $n = m + r$). n-bitowa jednostka zawierające bity danych i kontrolne jest często nazywana n-bitowym **słowem kodowym** (ang. *codeword*).

Mając dwa dowolne słowa kodowe, np. 10001001 i 10110001, możemy określić, ile odpowiadających sobie bitów różni się. W tym przypadku różnią się 3 bity. Aby ustalić liczbę odmiennych bitów, wystarczy wykonać operację XOR na dwóch słowach kodowych i zliczyć bity „1” w wyniku, na przykład:

```
 10001001
 10110001
 --------
 00111000
```

Liczba pozycji, na których różnią się słowa kodowe, nosi nazwę **odległości Hamminga** (Hamming, 1950). Jej znaczenie polega na tym, że jeśli dwa słowa kodowe są odległe o d bitów, to będą wymagać d jednobitowych błędów, aby przekształcić jedno w drugie.

W większości zastosowań transmisji danych wszystkie 2^m możliwe wersje komunikatu danych są legalne, lecz z uwagi na sposób, w jaki obliczane są bity kontrolne, nie wszystkie możliwe słowa kodowe (w liczbie 2^n) są używane. Na podstawie algorytmu obliczania bitów kontrolnych możemy teoretycznie zbudować pełną listę legalnych słów kodowych, a na tej liście znaleźć parę słów kodowych, dla której odległość Hamminga będzie minimalna. Będzie ona odległością Hamminga kompletnego kodu.

Zdolności wykrywania i korekty błędów przez kod zależą od jego odległości Hamminga. Aby wykryć d błędów, potrzebujemy kodu o odległości $d + 1$, ponieważ w takim kodzie d jednobitowych błędów nie będzie mogło w żaden sposób przekształcić poprawnego słowa kodowego w inne poprawne słowo kodowe. Gdy odbiornik otrzyma niepoprawne słowo kodowe, to będzie mógł stwierdzić wystąpienie błędu transmisji. Analogicznie, aby naprawić d błędów, potrzebujemy kodu o odległości $2d + 1$, ponieważ dzięki temu legalne słowa kodowe będą od siebie na tyle odległe, że nawet po d zmian oryginalne słowo kodowe będzie bliżej otrzymanego niż którekolwiek inne, co pozwoli je jednoznacznie zidentyfikować.

Jako prosty przykład kodu detekcyjnego rozważmy kod, w którym do danych dołączany jest pojedynczy **bit parzystości**. Ten bit jest wybierany tak, że liczba jedynek w słowie kodowym jest parzysta (lub nieparzysta). Na przykład, przy wysyłaniu 1011010 z bitem parzystości, na koniec dodawany jest bit dający w sumie ciąg 10110100. Z bitem nieparzystości z 1011010 otrzymamy 10110101. Kod z jednym bitem parzystości ma odległość 2, ponieważ dowolny pojedynczy błąd da słowo kluczowe z niewłaściwą parzystością. Ten kod pozwala wykrywać pojedyncze błędy.
Jako prosty przykład kodu korekcyjnego rozważmy kod mający tylko cztery poprawne słowa:

0000000000, 0000011111, 1111100000 i 1111111111

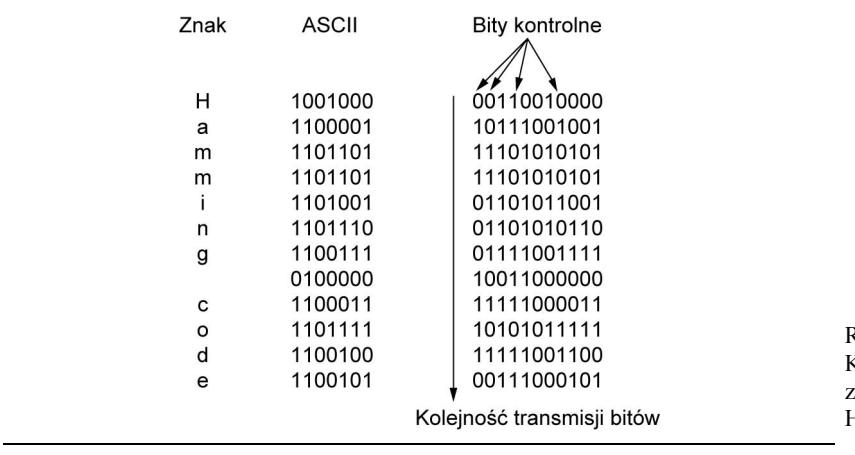
Odległość dla tego kodu wynosi 5, co oznacza, że może on naprawiać podwójne błędy. Jeśli zostanie odebrane słowo kodowe 0000000111, odbiornik rozpozna, że oryginałem musi być 0000011111. Jeśli jednak potrójny błąd zmieni 0000000000 na 0000000111, ten błąd nie zostanie poprawnie skorygowany.

Wyobraźmy sobie, że chcemy zaprojektować kod z m bitów komunikatu i r bitów kontrolnych pozwalający na korektę wszystkich pojedynczych błędów. Każdy z 2^m legalnych komunikatów ma n nielegalnych słów kodowych w odległości 1. Otrzymujemy je przez systematyczne negowanie każdego z n bitów w n-bitowym słowie kodowym utworzonym z komunikatu. Wobec tego każdy z 2^m legalnych komunikatów wymaga poświęcenia mu $n + 1$ wzorców bitów. Ponieważ całkowita liczba wzorców wynosi 2^n, musimy mieć $(n + 1)2^m \leq 2^n$. Używając $n = m + r$ otrzymujemy wymóg $(m + r + 1) \leq 2^r$. Znając m, możemy ustalić minimalną liczbę bitów kontrolnych niezbędną do korekty pojedynczych błędów.

Ten teoretyczny dolny limit jest w rzeczy samej do osiągnięcia za pomocą metody autorstwa Hamminga (1950). Bity słowa kluczowego są numerowane kolejno, zaczynając od 1 na lewym końcu, 2 zaraz po nim i tak dalej. Pozycje stanowiące potęgę liczby 2 (1, 2, 4, 8, 16 itd.) są bitami kontrolnymi. Pozostałe (3, 5, 6, 7, 9 itd.) są wypełniane m bitów danych. Każdy bit kontrolny wymusza parzystość (lub nieparzystość) pewnego zbioru bitów wraz z sobą. Bit może być zawarty w kilku obliczeniach parzystości. Aby zobaczyć, w których bitach kontrolnych uczestniczy bit na pozycji k, możemy rozłożyć k na sumę potęg liczby 2. Na przykład, $11 = 1 + 2 + 8$, a $29 = 1 + 4 + 8 + 16$. Bit jest sprawdzany tylko przez te bity kontrolne, które występują w tym rozwinięciu (np. bit 11 jest kontrolowany przez bity 1, 2 i 8).

Gdy zostaje odebrane słowo kluczowe, odbiornik zeruje licznik. Następnie sprawdza wszystkie bity kontrolne k ($k = 1, 2, 4, 8\ldots$), czy mają poprawną parzystość. Jeśli nie, odbiornik dodaje k do licznika. Jeśli po sprawdzeniu wszystkich bitów kontrolnych wartość w liczniku wynosi zero (tzn.

jeśli wszystkie były poprawne), to słowo kodowe jest przyjmowane jako poprawne. Jeśli licznik jest niezerowy, to zawiera numer błędnego bitu. Na przykład, jeśli bity kontrolne 1, 2 i 8 są błędne, to zanegowanym bitem jest nr 11, ponieważ jest to jedyny bit sprawdzany przez bity 1, 2 i 8. Rysunek 3.7 przedstawia kilka znaków ASCII zakodowanych jako 11-bitowe słowa kodowe z użyciem kodu Hamminga. Proszę pamiętać, że dane znajdują się na pozycjach 3, 5, 6, 7, 9, 10 i 11.



RYSUNEK 3.7. Korekta błędów za pomocą kodu Hamminga

Kody Hamminga mogą korygować tylko pojedyncze błędy. Jest jednak sztuczka, która pozwala na korektę przez te kody paczek błędów. Sekwencja *k* kolejnych słów kodowych jest układana w postaci tablicy, po jednym słowie kodowym na wiersz. Standardowo dane byłyby wysyłane jedno słowo kodowe po drugim, z lewej do prawej. Aby korygować błędy występujące w seriach, dane wysyła się kolumna po kolumnie, zaczynając od pierwszej kolumny z lewej. Po wysłaniu wszystkich *k* bitów zostaje wysłana następna kolumna i tak dalej, jak na rysunku 3.7. Gdy ramka dociera do odbiornika, macierz jest rekonstruowana kolumna po kolumnie. Jeśli wystąpi paczka błędów o długości *k*, to wpłynie to najwyżej na 1 bit w każdym słowie kodowym. Ponieważ kod Hamminga potrafi korygować jeden błąd w słowie kluczowym, to będzie można przywrócić cały blok. Ta metoda stosuje *kr* bitów kontrolnych, aby uniewrażliwić blok *km* bitów danych na pojedynczą paczkę błędów o długości *k* lub mniejszej.

3.2.2. Kody detekcyjne

Kody korekcyjne są powszechnie stosowane w łączach bezprzewodowych cieszących się złą sławą z powodu zakłóceń i podatności na błędy w porównaniu z kablami miedzianymi i światłowodami. Bez kodów korekcyjnych trudno byłoby przesłać przez nie cokolwiek. Jednakże w kablach miedzianych i światłowodach stopa błędów jest znacznie niższa, więc wykrywanie błędów i retransmisja jest w nich zwykle bardziej wydajna przy radzeniu sobie z okazyjnymi błędami.

Jako prosty przykład rozważmy kanał, w którym błędy są odosobnione, a stopa błędów wynosi 10^{-6} na bit. Niech blok ma rozmiar 1000 bitów. Aby zapewnić korektę błędów dla bloków o tej wielkości potrzebnych jest 10 bitów kontrolnych; megabit danych wymagałby 10 000 bitów kontrolnych. Aby jedynie wykryć blok z jednobitowym błędem, wystarczy jeden bit parzystości na blok. Raz na 1000 bloków trzeba będzie nadać dodatkowy blok (1001 bitów). Całkowite dodatkowe obciążenie kanału w metodzie detekcji błędów i retransmisji wynosi tylko 2001 bitów na megabit danych, przy 10 000 bitów dla kodu Hamminga.

Jeśli do bloku dodany jest pojedynczy bit parzystości, a blok zostanie poważnie uszkodzony przez długą paczkę błędów, to prawdopodobieństwo wykrycia błędów wyniesie tylko 0,5, co jest raczej nie do przyjęcia. Szansę tę możemy znacząco poprawić, jeśli każdy blok przeznaczony do wysłania uznamy za tablicę o szerokości n i wysokości k bitów, jak w poprzednim punkcie. Dla każdej kolumny obliczymy osobny bit parzystości i dołączymy do tablicy jako ostatni wiersz. Następnie tablica zostanie wysłana wiersz po wierszu. Gdy blok dotrze na miejsce, odbiornik sprawdzi wszystkie bity parzystości. Jeśli którykolwiek z nich będzie niewłaściwy, odbiornik zażąda ponownej transmisji całego bloku. Żądania dodatkowych retransmisji są zgłaszane w miarę potrzeb, dopóki cały blok nie zostanie odebrany bez żadnych błędów parzystości.

Ta metoda pozwala wykryć pojedynczą paczkę błędów o długości n, ponieważ tylko 1 bit na kolumnę ulegnie zmianie. Paczka o długości $n + 1$ przejdzie jednak niewykryta, jeśli pierwszy bit zostanie zanegowany, ostatni zanegowany, a wszystkie pozostałe będą poprawne (wystąpienie paczki błędów nie oznacza, że wszystkie bity są przekłamane; oznacza jedynie, że błędne są pierwszy i ostatni bit). Jeśli blok zostanie poważnie uszkodzony przez jedną długą paczkę błędów lub przez kilka krótszych, prawdopodobieństwo, że dla dowolnej z n kolumn parzystość będzie przypadkowo poprawna, wynosi 0,5, więc prawdopodobieństwo zaakceptowania uszkodzonego bloku jako poprawnego wynosi 2^{-n}.

Wprawdzie powyższy schemat może być czasem wystarczający, lecz w praktyce powszechnie stosuje się inną metodę: **kod wielomianowy**, znany również pod nazwą **CRC** (**Cyclic Redundancy Check** — kontrola redundancji cyklicznej). Kody wielomianowe opierają się na traktowaniu łańcuchów bitów jako reprezentacji wielomianów ze współczynnikami wynoszącymi tylko 0 lub 1. Ramka o długości k bitów jest traktowana jak lista współczynników dla k-składnikowego wielomianu w zakresie od x^{k-1} do x^0. Mówimy, że taki wielomian jest stopnia $n - 1$. Najbardziej znaczący bit (pierwszy z lewej) jest współczynnikiem dla x^{k-1}; następny współczynnikiem dla x^{k-2} i tak dalej. Na przykład 110001 ma sześć bitów, więc reprezentuje sześcioskładnikowy wielomian ze współczynnikami 1, 1, 0, 0, 0 i 1: $x^5 + x^4 + x^0$.

Arytmetyka wielomianów jest przeprowadzana modulo 2, zgodnie z regułami teorii ciał algebraicznych. Nie ma żadnych przeniesień do dodania ani pożyczek do odjęcia. Zarówno dodawanie, jak i odejmowanie jest identyczne jak XOR. Na przykład:

```
  10011011   00110011   11110000   01010101
 +11001010  +11001101  -10100110  -10101111
 ________   ________   ________   ________
  01010001   11111110   01010110   11111010
```

Dzielenie przez liczby wielocyfrowe przeprowadza się tak samo jak w systemie binarnym, z tym wyjątkiem, że odejmowanie odbywa się modulo 2, jak powyżej. Mówimy, że dzielnik „mieści się" w dzielnej, jeśli dzielna ma przynajmniej tyle bitów co dzielnik.

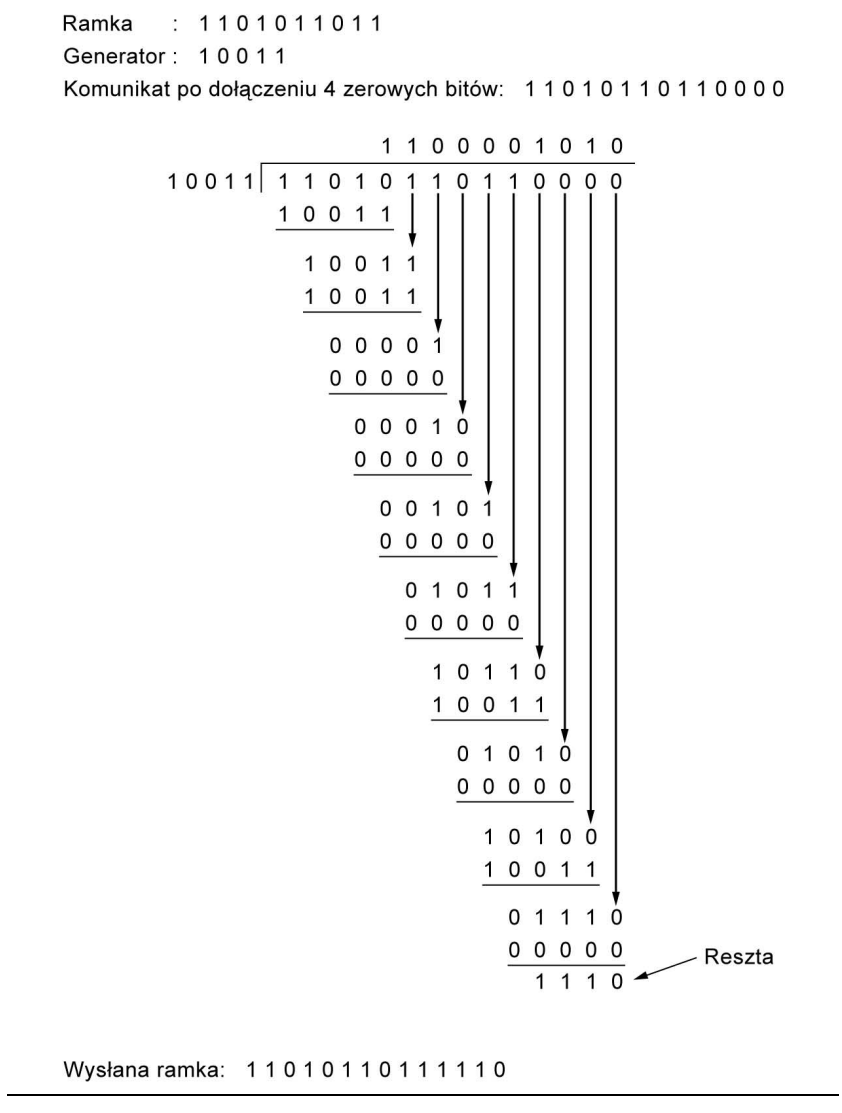
Gdy stosowana jest metoda kodów wielomianowych, nadajnik i odbiornik muszą z góry uzgodnić między sobą **wielomian generujący** $G(x)$. Zarówno bardziej, jak i mniej znaczące bity tego wielomianu muszą mieć wartość 1. Aby obliczyć **sumę kontrolną** dla jakiejś ramki o długości m bitów odpowiadającej wielomianowi $M(x)$, ramka musi być dłuższa od wielomianu generującego. Cały pomysł polega na dołączeniu sumy kontrolnej do końca ramki w ten sposób, że wielomian reprezentowany przez ramkę z sumą kontrolną jest podzielny przez $G(x)$. Gdy odbiornik otrzymuje taką ramkę z sumą kontrolną, to próbuje ją podzielić przez $G(x)$. Jeśli pozostanie reszta, oznacza to, że wystąpił błąd transmisji.

Algorytm obliczania sumy kontrolnej jest następujący:

(1) Niech r będzie stopniem wielomianu $G(x)$. Dołącz r zerowych bitów do mniej znaczącego końca ramki tak, że będzie zawierał $m + r$ bitów i odpowiadał wielomianowi $x^r M(x)$.

(2) Podziel łańcuch bitów odpowiadający $G(x)$ przez łańcuch $x^r M(x)$, używając dzielenia modulo 2.
(3) Odejmij resztę (która ma zawsze r lub mniej bitów) od łańcucha bitów $x^r M(x)$, używając odejmowania modulo 2. Wynikiem jest ramka z sumą kontrolną przeznaczona do wysłania. Nazwijmy ją wielomianem $T(x)$.

Rysunek 3.8 ilustruje obliczenia dla ramki 1101011011 z użyciem wielomianu generującego $G(x) = x^4 + x + 1$.



RYSUNEK 3.8. Obliczanie sumy kontrolnej w kodzie wielomianowym

Powinno być jasne, że wielomian $T(x)$ jest podzielny (modulo 2) przez $G(x)$. W każdym problemie dzielenia, jeśli zmniejszymy dzielną o resztę, to pozostałość będzie podzielna przez dzielnik. Na przykład, jeśli w systemie dziesiętnym podzielimy 210 278 przez 10 941, pozostanie reszta równa 2399. Gdy odejmiemy 2399 od 210 278, to pozostałość (207 879) będzie podzielna przez 10 941.

Przeanalizujmy teraz możliwości tej metody. Jakiego typu błędy zostaną wykryte? Załóżmy, że wystąpi błąd transmisji, przez który zamiast ciągu bitów $T(x)$ zostanie odebrany $T(x) + E(x)$. Każdy bit równy 1 w $E(x)$ odpowiada bitowi, który został zanegowany w oryginale. Jeśli $E(x)$ zawiera k jedynek, to wystąpiło k jednobitowych błędów. Pojedyncza paczka błędów jest identyfikowana przez 1 na początku, a kombinację zer i jedynek i jedynkę na końcu; pozostałe bity są równe zero.

Po odebraniu ramki z sumą kontrolną odbiornik dzieli ją przez $G(x)$; inaczej mówiąc, oblicza $[T(x) + E(x)]/G(x)$. $T(x)/G(x)$ jest równe 0, więc wynikiem obliczenia jest po prostu $E(x)/G(x)$. Błędy, które odpowiadają wielomianowi zawierającemu $G(x)$ jako czynnik prześlizgną się; wszystkie pozostałe zostaną wychwycone.

Jeśli wystąpił błąd na jednym bicie, to $E(x) = x^i$, gdzie i określa, który bit jest błędny. Jeśli $G(x)$ zawiera dwa lub więcej składników, to nigdy nie podzieli $E(x)$, więc wszystkie jednobitowe błędy zostaną wykryte.

Jeśli wystąpią dwa odizolowane błędy jednobitowe, to $E(x) = x^i + x^j$, gdzie $i > j$. Alternatywnie można to zapisać w postaci to $E(x) = x^j(x^{i-j} + 1)$. Jeśli założymy, że $G(x)$ nie jest podzielny przez x, to wystarczającym warunkiem do wykrycia wszystkich błędów jest to, że $G(x)$ nie będzie dzielić $x^k + 1$ dla każdego k aż do maksymalnej wartości $i – j$ (tzn. aż do maksymalnej długości ramki). Znane są proste wielomiany niskiego stopnia, które zapewniają ochronę długich ramek. Na przykład, $x^{15} + x^{14} + 1$ nie podzieli $x^k + 1$ dla żadnej wartości k poniżej 32 768.

Jeśli błędna jest nieparzysta liczba bitów, $E(x)$ zawiera nieparzystą liczbę składników (np. $x^5 + x^2 + 1$, lecz nie $x^2 + 1$). Co ciekawe, żaden wielomian z nieparzystą liczbą składników nie zawiera w systemie modulo 2 czynnika $x + 1$. Wybierając $x + 1$ jako czynnik $G(x)$, możemy wyłapać wszystkie błędy składające się z nieparzystej liczby zanegowanych bitów.

Aby udowodnić, że żaden wielomian z nieparzystą liczbą składników nie jest podzielny przez $x + 1$, załóżmy, że $E(x)$ ma nieparzystą liczbę składników i jest podzielny przez $x + 1$. Rozłóżmy $E(x)$ na $(x + 1)Q(x)$. Teraz obliczmy $E(1) = (1+1)Q(1)$. Ponieważ $1 + 1 = 0$ (w modulo 2), to $E(1)$ musi równać się zero. Jeśli $E(x)$ ma nieparzystą liczbę składników, to zastąpienie wszędzie x wartością 1 zawsze zwróci w wyniku 1. Wobec tego żaden wielomian z nieparzystą liczbą składników nie jest podzielny przez $x + 1$.

Na koniec, co najważniejsze, kod wielomianowy z r bitów kontrolnych wykrywa każdą paczkę błędów o długości mniejszej lub równej r. Paczkę błędów możemy przedstawić w postaci $x^i(x^{k-1} + ... + 1)$, gdzie i określa, jak daleko od prawego końca odebranej ramki mieści się paczka błędów. Jeśli $G(x)$ zawiera składnik x^0, to nie będzie miał czynnika x^i, jeśli więc stopień wyrażenia w nawiasach jest mniejszy od stopnia $G(x)$, to reszta nigdy nie będzie równa zero.

Jeśli paczka ma długość $r + 1$, to reszta z dzielenia przez $G(x)$ będzie równa zero wtedy i tylko wtedy, gdy paczka będzie identyczna z $G(x)$. Z definicji pierwszy i ostatni bit paczki muszą być równe 1, więc dopasowanie zależy od $r – 1$ pośrednich bitów. Jeśli uznamy wszystkie kombinacje za jednakowo prawdopodobne, to prawdopodobieństwo uznania takiej błędnej ramki za poprawną będzie wynosić $(1/2)^{r-1}$.

Można też pokazać, że gdy wystąpi paczka błędów dłuższa niż $r + 1$ bitów lub gdy wystąpi kilka krótszych paczek, to prawdopodobieństwo przejścia niezauważonej błędniej ramki wyniesie $(1/2)^r$, zakładając, że wszystkie wzorce bitów mają takie samo prawdopodobieństwo.

Niektóre wielomiany stały się międzynarodowymi standardami. W IEEE 802 używany jest:

$$x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + x^1 + 1$$

Poza innymi pożądanymi właściwościami wykrywa wszystkie paczki błędów o długości 32 i mniej bitów oraz wszystkie paczki zmieniające nieparzystą liczbę bitów.

Wprawdzie obliczenia wymagane do uzyskania sumy kontrolnej mogą wydawać się skomplikowane, to Peterson i Brown (1961) pokazali, że można sprzętowo skonstruować prosty układ rejestru przesuwnego do obliczania i weryfikacji sum kontrolnych. W praktyce niemal zawsze stosuje się takie urządzenie: praktycznie we wszystkich sieciach LAN i w niektórych przypadkach na liniach dwupunktowych.

Przez dziesiątki lat zakładano, że ramki przeznaczone do objęcia sumą kontrolną zawierają losowe bity. Wszystkie analizy algorytmów sum kontrolnych przeprowadzano z takim założeniem. Badania rzeczywistych danych pokazały, że jest ono całkowicie mylne. Na skutek tego w pewnych warunkach niewykryte błędy występują znacznie częściej, niż uprzednio sądzono (Partridge i in., 1995).

3.3. Podstawowe protokoły łącza danych

Omawianie protokołów zaczniemy od opisu trzech protokołów o rosnącej złożoności. Dla zainteresowanych dostępny jest poprzez WWW symulator tych i kolejnych protokołów (patrz Wprowadzenie). Zanim przyjrzymy się tym protokołom, warto sprecyzować kilka założeń dotyczących modelu komunikacji, na którym się opierają. Zacznijmy od tego, że zakładamy obecność w warstwach fizycznej, łącza danych i sieciowej niezależnych procesów, które komunikują się za pomocą przekazywanych tam i z powrotem komunikatów. W wielu przypadkach procesy warstw fizycznej i łącza danych będą uruchomione w procesorze wewnątrz specjalnego układu wejścia-wyjścia, a kod warstwy sieciowej będzie uruchomiony w głównym procesorze komputera. Jednakże możliwe są też inne implementacje (np. trzy procesy w jednym układzie scalonym wejścia-wyjścia lub warstwy fizyczna i łącza danych jako procedury wywoływane przez proces warstwy sieciowej). W każdym razie traktowanie trzech warstw jako odrębnych procesów spowoduje, że opisy będą ideowo bardziej przejrzyste, a oprócz tego pomoże w podkreśleniu niezależności warstw.

Kolejnym podstawowym założeniem jest to, że komputer *A* chce wysłać długi strumień danych do komputera *B* z użyciem niezawodnej usługi połączeniowej. Później rozważymy też przypadek, gdy *B* też chce równocześnie wysyłać dane do *A*. Zakładamy, że *A* ma nieskończone zasoby danych gotowych do wysłania i nigdy nie musi czekać na ich wygenerowanie. Zamiast tego, gdy warstwa łącza danych w *A* żąda danych, warstwa sieciowa jest zawsze gotowa natychmiast je dostarczyć (to ograniczenie również zarzucimy później).

Zakładamy też, że komputery nie padają. Inaczej mówiąc, protokoły radzą sobie z błędami komunikacji, lecz nie z problemami powodowanymi zawieszaniem i restartem komputera.

Z punktu widzenia warstwy łącza danych pakiet przekazywany do niej przez interfejs z warstwy sieciowej stanowi czyste dane, których każdy bit musi zostać doręczony do warstwy sieciowej w komputerze docelowym. Fakt, że ta warstwa może potraktować część pakietu jako nagłówek, nie jest istotny dla warstwy łącza danych.

Gdy warstwa łącza danych przyjmuje pakiet, kapsułkuje go w ramkę przez dodanie do niego nagłówka i stopki łącza danych (patrz rysunek 3.1). Wobec tego ramka składa się z osadzonego w niej pakietu, jakichś informacji sterujących (w nagłówku) i z sumy kontrolnej (w stopce). Następnie ramka zostaje wysłana do warstwy łącza danych w drugim komputerze. Zakładamy, że istnieją odpowiednie procedury biblioteczne: `to_physical_layer`, służąca do wysłania ramki, i `from_physical_layer`, służąca do odebrania ramki. Sprzęt nadajnika oblicza i dołącza sumę kontrolną (tworząc w ten sposób stopkę), więc oprogramowanie warstwy łącza danych nie musi się tym zajmować. Może tu zostać użyty, na przykład, algorytm wielomianowy omówiony wcześniej.

Na początku odbiornik nie ma nic do roboty. Czeka tylko, aż coś się wydarzy. W przykładowych protokołach w niniejszym rozdziale oznaczymy, że warstwa łącza danych czeka na jakieś zdarzenie, przez wywołanie procedury `wait_for_event(&event)`. Powrót z tej procedury odbywa się tylko wtedy, gdy coś się stanie (np. nadejdzie ramka). Po powrocie zmienna `event` informuje, co się stało. Zestaw możliwych zdarzeń będzie różny w różnych opisywanych protokołach i zdefiniujemy go osobno dla każdego protokołu. Proszę zwrócić uwagę, że w bardziej realistycznej sytuacji warstwa łącza danych nie będzie czekać na zdarzenie w ciasnej pętli, jak zasugerowaliśmy, lecz otrzyma przerwanie, które spowoduje wstrzymanie aktualnego działania i przejście do obsługi przychodzącej ramki. Jednakże dla uproszczenia zignorujemy wszystkie szczegóły równoległych działań w warstwie łącza danych i założymy, że cały swój czas poświęca na obsługę tylko naszego jednego kanału.

Gdy ramka dociera do odbiornika, sprzęt oblicza sumę kontrolną. Jeśli suma kontrolna jest błędna (tzn. wystąpił błąd transmisji), zostaje o tym poinformowana warstwa łącza danych (`event = cksum_err`). Jeśli przychodząca ramka dotarła bez uszkodzeń, również informowana jest warstwa łącza danych (`event = frame_arrival`), aby mogła pobrać ramkę do analizy za pomocą `from_physical_layer`. Warstwa łącza danych w odbiorniku zaraz po otrzymaniu nieuszkodzonej ramki sprawdza informacje sterujące w jej nagłówku, i jeśli wszystko jest w porządku, przesyła zawarty w niej pakiet do warstwy sieciowej. Nagłówek ramki absolutnie nigdy nie jest przekazywany do warstwy sieciowej.

Istnieje dobry powód, dla którego warstwa sieciowa nie może nigdy otrzymać jakiejkolwiek części nagłówka ramki — aby utrzymać całkowitą separację protokołów sieciowych i łącza danych. Dopóki warstwa sieciowa nie wie nic o protokole łącza danych i o formacie ramki, można je zmieniać bez konieczności zmian w oprogramowaniu warstwy sieciowej. Ustalenie sztywnego interfejsu pomiędzy warstwami sieciową i łącza danych ogromnie upraszcza tworzenie oprogramowania, ponieważ protokoły komunikacyjne w różnych warstwach mogą ewoluować niezależnie od siebie.

Listing 3.1 przedstawia kilka deklaracji (w języku C), wspólnych dla wielu protokołów, które omówimy później. Zostało tu zdefiniowanych pięć struktur danych: `boolean`, `seq_nr`, `packet`, `frame_kind` i `frame`. `boolean` jest typem wyliczeniowym i może przyjmować wartości `true` i `false`. `seq_nr` jest małą liczbą całkowitą (*small integer*) służącą do numerowania ramek, co pozwoli je odróżnić od siebie. Te numery sekwencyjne mają wartości od 0 do maksymalnej `MAX_SEQ`, która jest zdefiniowana w każdym potrzebującym tego protokole. `packet` jest jednostką informacji wymienianą pomiędzy warstwą sieciową i warstwą łącza danych w tym samym komputerze lub pomiędzy równoległymi warstwami sieciowymi. W naszym modelu pakiet zawsze ma `MAX_PKT` bajtów, lecz bardziej realistyczna byłaby zmienna długość.

LISTING 3.1.
Definicje potrzebne w protokołach opisanych w dalszej części rozdziału.
Definicje te mieszczą się w pliku protocol.h

```
#define MAX_PKT 1024                          /* ustala wielkość pakietu w bajtach */

typedef enum {false, true} boolean;           /* typ boolean */
typedef unsigned int seq_nr;                  /* Numery sekwencyjne lub ack */
typedef struct {unsigned char data[MAX_PKT];} packet;   /* definicja pakietu (packet) */
typedef enum {data, ack, nak} frame_kind;     /* definicja typu ramki (frame_kind) */

typedef struct {                              /* W tej warstwie są przenoszone ramki */
   frame_kind kind;                           /* co to za typ ramki? */
   seq_nr seq;                                /* numer sekwencyjny */
   seq_nr ack;                                /* numer potwierdzenia */
   packet info;                               /* pakiet warstwy sieciowej */
} frame;

/* Czekaj na zdarzenie; zwróć jego typ w event. */
void wait_for_event(event_type *event);

/* Pobierz z warstwy sieciowej pakiet do przesłania kanałem. */
void from_network_layer(packet *p);

/* Doręcz informacje z przychodzącej ramki do warstwy sieciowej. */
void to_network_layer(packet *p);
```

```
/* Pobierz ramkę przychodzącą z warstwy fizycznej i skopiuj ją do r. */
void from_physical_layer(frame *r);

/* Przekaż ramkę do warstwy fizycznej w celu transmisji. */
void to_physical_layer(frame *s);

/* Uruchom zegar i aktywuj zdarzenie timeout. */
void start_timer(seq_nr k);

/* Zatrzymaj zegar i dezaktywuj zdarzenie timeout. */
void stop_timer(seq_nr k);

/* Uruchom czasomierz pomocniczy i aktywuj zdarzenie ack_timeout. */
void start_ack_timer(void);

/* Zatrzymaj czasomierz pomocniczy i dezaktywuj zdarzenie ack_timeout. */
void stop_ack_timer(void);

/* Pozwól warstwie sieciowej powodować zdarzenie network_layer_ready. */
void enable_network_layer(void);

/* Zabroń warstwie sieciowej powodować zdarzenie network_layer_ready. */
void disable_network_layer(void);

/* Makro inc jest wprost podmieniane w kodzie przed kompilacją: inkrementuj k
cyklicznie. */
#define inc(k) if (k < MAX_SEQ) k = k + 1; else k = 0
```

Ramka (`frame`) składa się z czterech pól: `kind`, `seq`, `ack` i `info`, z których trzy pierwsze zawierają informacje kontrolne, a ostatnie może zawierać faktyczne dane do przesłania. Pola kontrolne łącznie noszą nazwę **nagłówka ramki**.

Pole `kind` informuje, czy ramka zawiera jakieś dane, ponieważ niektóre protokoły odróżniają ramki zawierające tylko informacje sterujące od ramek, które zawierają też dane. Pola `seq` i `ack` mieszczą odpowiednio numery sekwencyjne i potwierdzenia; ich zastosowanie opiszemy bardziej szczegółowo później. Pole `info` ramki danych zawiera jeden pakiet; w ramce sterującej nie jest używane. Bardziej realistyczna implementacja używałaby pola `info` o zmiennej długości, pomijając je całkowicie w ramkach sterujących.

Ponownie musimy zdać sobie sprawę z relacji pomiędzy pakietem i ramką. Warstwa sieciowa buduje pakiet, biorąc wiadomość z warstwy transportowej i dodając do niej nagłówek warstwy sieciowej. Ten pakiet jest przesyłany do warstwy łącza danych w celu zawarcia w polu `info` wysyłanej ramki. Gdy ramka dociera do miejsca przeznaczenia, warstwa łącza danych wydobywa pakiet z ramki i przekazuje go do warstwy sieciowej. W ten sposób warstwa sieciowa może działać tak, jakby komputery mogły bezpośrednio przesyłać pomiędzy sobą ramki.

Listing 3.1 wymienia również szereg procedur. Są to procedury biblioteczne, których szczegóły zależą od implementacji, a ich wewnętrzne funkcjonowanie nie będzie nas tu więcej zajmować. Procedura `wait_for_event` czeka w ciasnej pętli na jakieś zdarzenie, jak już wspomnieliśmy. Procedury `to_network_layer` i `from_network_layer` są używane przez warstwę łącza danych odpowiednio do przekazywania pakietów do warstwy sieciowej i przyjmowania pakietów z tej warstwy. Proszę zwrócić uwagę, że `from_physical_layer` i `to_physical_layer` przekazują ramki pomiędzy warstwami fizyczną i łącza danych. Z drugiej strony procedury `to_network_layer` i `from_network_layer` przekazują pakiety pomiędzy warstwami łącza danych i sieciową. Inaczej mówiąc, `to_network_layer` i `from_network_layer` są związane z interfejsem pomiędzy warstwami 2. i 3., a `from_physical_layer` i `to_physical_layer` pomiędzy warstwami 1. i 2.

W większości protokołów zakładamy, że kanał nie jest niezawodny i od czasu do czasu traci całe ramki. Aby móc przywrócić łączność po takich katastrofach, nadająca warstwa łącza danych musi uruchomić wewnętrzny czasomierz lub zegar za każdym razem, gdy wysyła ramkę. Jeśli w określonym z góry czasie nie zostanie odebrana odpowiedź, upłynie interwał odliczany przez zegar i warstwa łącza danych odbierze sygnał przerwania.

W naszych protokołach jest to obsługiwane przez zezwolenie procedurze `wait_for_event` na zwrócenie `event = timeout`. Procedury `start_timer` i `stop_timer` odpowiednio włączają i wyłączają czasomierz. Upłynięcie czasu oczekiwania jest możliwe tylko wtedy, gdy czasomierz jest uruchomiony; takie wywołanie po prostu zeruje zegar, aby następne upłynięcie czasu oczekiwania nastąpiło po pełnym interwale czasomierza (o ile nie zostanie w tym czasie wyzerowany lub wyłączony).

Procedury `start_ack_timer` i `stop_ack_timer` sterują czasomierzem pomocniczym, który służy do generowania potwierdzeń w określonych warunkach.

Procedury `enable_network_layer` i `disable_network_layer` są używane w bardziej wyrafinowanych protokołach, w których nie zakładamy już, że warstwa sieciowa zawsze ma pakiety do wysłania. Gdy warstwa łącza danych aktywuje warstwę sieciową, wtedy warstwa sieciowa ma prawo wysłać przerwanie, jeśli ma pakiet do wysłania. Wskazujemy to przez `event = network_layer_ready`. Gdy warstwa sieciowa jest nieaktywna, nie może powodować takich zdarzeń. Warstwa łącza danych przez odpowiednie włączanie i wyłączanie swojej warstwy sieciowej może chronić przed zalaniem ze strony warstwy sieciowej pakietami, na które nie ma miejsca w buforach.

Numery sekwencyjne ramek zawsze mieszczą się w przedziale od 0 do `MAX_SEQ` (włącznie), gdzie `MAX_SEQ` może mieć różne wartości dla różnych protokołów. Często konieczne jest zwiększanie numeru sekwencyjnego o 1 cyklicznie (tzn. po `MAX_SEQ` następuje 0). Inkrementację przeprowadza makro `inc`. Operacja ta została zdefiniowana jako makro, ponieważ jest wstawiana do ścieżki krytycznej. Jak zobaczymy później, czynnikiem ograniczającym wydajność sieci jest często przetwarzanie protokołu, więc definiowanie prostych operacji jak ta w postaci makr nie pogarsza czytelności kodu, lecz poprawia wydajność. Ponadto, ponieważ `MAX_SEQ` ma różne wartości w różnych protokołach, to w postaci makra operację możemy włączyć do wszystkich protokołów w tym samym programie binarnym bez konfliktu. Ta zdolność jest przydatna w symulatorze.

Deklaracje z listingu 3.1 należą do wszystkich opisanych poniżej protokołów. Aby zaoszczędzić miejsca i udostępnić praktyczne odniesienie, zostały one wydobyte i zgrupowane razem, lecz ideowo powinny być scalone z samymi protokołami. W C takie scalanie odbywa się przez umieszczenie definicji w specjalnym pliku nagłówkowym, w naszym przypadku *protocol.h*, i użycie `#include` z preprocesora C w celu zawarcia ich w plikach protokołów.

3.3.1. Nieograniczony protokół simpleksowy

Na pierwszy ogień weźmiemy protokół najprostszy z możliwych. Dane przesyłane są tylko w jednym kierunku. Warstwy sieciowe nadająca i odbierająca są zawsze gotowe. Czas przetwarzanie można pominąć. Dostępne jest nieograniczone miejsce w buforach. I, co najlepsze, kanał komunikacyjny pomiędzy warstwami łącza danych nigdy nie uszkadza ani nie traci ramek. Ten zupełnie oderwany od rzeczywistości protokół, który nazwiemy „utopia", przedstawia listing 3.2.

LISTING 3.2.
Nieograniczony protokół simpleksowy

```
/* Protokół 1. (utopia) umożliwia transmisję danych tylko w jednym kierunku,
   z nadajnika do odbiornika. Zakładamy, że kanał komunikacyjny jest wolny
   od błędów, a odbiornik jest w stanie przetwarzać wszystkie dane wejściowe
```

```
   nieskończenie szybko. Wobec tego nadajnik po prostu tkwi w pętli
   i wysyła dane w linię tak szybko, jak potrafi. */

typedef enum {frame_arrival} event_type;
#include "protocol.h"

void sender1(void)
{
   frame s;                          /* bufor dla wychodzącej ramki */
   packet buffer;                    /* bufor dla wychodzącego pakietu */

   while (true) {
      from_network_layer(&buffer);   /* idź pobrać coś do wysłania */
      s.info = buffer;               /* skopiuj to do s w celu transmisji */
      to_physical_layer(&s);         /* wyślij to w drogę */
   }                                 /* Ciągle to jutro, jutro i znów jutro
                                        Wije się w ciasnym kółku od dnia do dnia
                                        Aż do ostatniej głoski czasokresu;
                                        - Makbet, V, v */
}

void receiver1(void)
{
   frame r;
   event_type event;                 /* wypełnione przez wait, lecz tu nieużywane */

   while (true) {
      wait_for_event(&event);        /* jedyna możliwość to frame_arrival */
      from_physical_layer(&r);       /* idź pobrać przychodzącą ramkę */
      to_network_layer(&r.info);     /* przekaż dane do warstwy sieciowej */
   }
}
```

Protokół ten składa się z dwóch odrębnych procedur: nadajnika i odbiornika. Nadajnik działa w warstwie łącza danych komputera źródłowego, a odbiornik w warstwie łącza danych komputera docelowego. Nie są tu używane żadne numery sekwencyjne, więc nie potrzeba `MAX_SEQ`. Jedynym możliwym typem zdarzenia jest `frame_arrival` (tzn. nadejście nieuszkodzonej ramki).

Nadajnik pracuje w nieskończonej pętli, po prostu wysyłając dane w linię tak szybko, jak potrafi. Na treść pętli składają się trzy czynności: pobranie pakietu z warstwy sieciowej (zawsze gotowej do tego), skonstruowanie wychodzącej ramki z użyciem zmiennej *s* i wysłanie ramki w drogę. Ten protokół wykorzystuje tylko pole `info` ramki, ponieważ pozostałe pola są związane z kontrolą błędów i sterowaniem przepływem, a w naszym modelu nie występują błędy i nie ma ograniczeń przepływu.

Odbiornik jest równie prosty. Na początku czeka na jakieś zdarzenie; jedyną możliwością jest przybycie nieuszkodzonej ramki. W końcu ramka dociera, co powoduje wyjście z procedury `wait_for_event` z wartością `frame_arrival` w `event` (która i tak jest ignorowana). Wywołanie `from_physical_layer` powoduje pobranie nadesłanej ramki z bufora sprzętowego i umieszczenie jej w zmiennej `r`, skąd może pobrać ramkę kod odbiornika. Na koniec sekcja danych jest przekazywana do warstwy sieciowej, a warstwa łącza danych wraca do oczekiwania na następną ramkę, praktycznie zawieszając się aż do nadejścia ramki.

3.3.2. Simpleksowy protokół stop-and-wait

Zrezygnujemy teraz z najmniej realistycznego założenia użytego w protokole 1. — zdolności warstwy sieciowej odbiornika do przetwarzania danych nieskończenie szybko (lub, co na to samo wychodzi, obecności nieskończonej ilości miejsca w buforze do przechowywania przychodzących ramek, podczas gdy będą czekać na swoją kolej). Nadal zakładamy, że kanał komunikacyjny jest wolny od błędów i że dane przesyłane są w jednym kierunku.

Główny problem, z którym musimy się tu uporać, polega na tym, aby zapobiec przed zalewaniem odbiornika przez nadajnik danymi szybciej, niż odbiornik będzie mógł je przetworzyć. W skrócie, jeśli odbiornik wymaga czasu Δt do wykonania operacji `from_physical_layer` plus `to_network_layer`, nadajnik musi wysyłać ramki z szybkością mniejszą niż jedna ramka na Δt. Co więcej, jeśli założymy, że w odbiorniku nie odbywa się sprzętowo automatyczne buforowanie i kolejkowanie, nadajnikowi nie wolno nadać nowej ramki, dopóki poprzednia nie zostanie pobrana przez `from_physical_layer`; w przeciwnym razie nowa ramka zamazałaby starą.

W niektórych ograniczonych przypadkach (np. w transmisji synchronicznej, gdzie warstwa łącza danych odbiornika jest w pełni poświęcona przetwarzaniu jednej linii wejściowej) może wystarczyć po prostu wstawienie w nadajniku opóźnienia w protokole 1. spowalniającego transmisję na tyle, że odbiornik nie zostanie zalany ramkami. Jednakże częściej każda warstwa łącza danych musi zajmować się kilkoma liniami, a odstępy czasu pomiędzy nadejściem ramki i jej przetworzeniem mogą mieć duży rozrzut. Jeśli projektanci sieci potrafią obliczyć zachowanie odbiornika w najbardziej niekorzystnych warunkach, będą mogli zaprogramować nadajnik do wysyłania informacji tak powoli, że nawet przy maksymalnym opóźnieniu ramki nie będzie przepełnień. Takie podejście jest jednak zbyt konserwatywne. Prowadzi do wykorzystania pasma znacznie poniżej optimum, chyba że wahania czasu reakcji warstwy łącza danych są bardzo małe (najlepsze i najgorsze warunki są niemal identyczne).

Bardziej ogólnym rozwiązaniem tego problemu jest dostarczanie przez odbiornik informacji z powrotem do nadajnika. Odbiornik po przekazaniu pakietu do swojej warstwy sieciowej odsyła do nadajnika małą pustą ramkę, która stanowi zezwolenie wysłania kolejnej ramki. Po wysłaniu ramki przez nadajnik protokół wymaga od niego oczekiwania na nadejście małej pustej ramki (potwierdzenia). Informowanie nadajnika przez odbiornik, kiedy może wysłać kolejną porcję danych, jest przykładem wspomnianego wcześniej sterowania przepływem.

Protokoły, w których nadajnik wysyła jedną ramkę i czeka na potwierdzenie zanim wyśle następną, noszą nazwę „**zatrzymaj się i czekaj**” (**stop-and-wait**). Listing 3.3 przedstawia przykład simpleksowego protokołu stop-and-wait.

LISTING 3.3.
Simpleksowy protokół stop-and-wait

```
/* Protokół 2. (stop-and-wait) również umożliwia transmisję danych w jednym kierunku
   z nadajnika do odbiornika. Ponownie zakładamy, że kanał komunikacyjny jest wolny
   od błędów jak w protokole 1. Jednakże tym razem odbiornik ma tylko skończoną
   pojemność buforów i skończoną szybkość przetwarzania, więc protokół musi wprost
   uniemożliwiać nadajnikowi zalewanie odbiornika danymi szybciej, niż mogą być
   przetwarzane. */

typedef enum {frame_arrival} event_type;
#include "protocol.h"

void sender2(void)
{
```

```
   frame s;                        /* bufor dla wychodzącej ramki */
   packet buffer;                  /* bufor dla wychodzącego pakietu */
   event_type event;               /* jedyna możliwość to frame_arrival */

   while (true) {
      from_network_layer(&buffer); /* idź pobrać coś do wysłania */
      s.info = buffer;             /* skopiuj to do s w celu transmisji */
      to_physical_layer(&s);       /* szczęśliwej podróży, rameczko */
      wait_for_event(&event);      /* nie idź dalej, zanim nie dostaniesz pozwolenia */
   }
}

void receiver2(void)
{
   frame r, s;                     /* bufory na ramki*/
   event_type event;               /* jedyna możliwość to frame_arrival */
   while (true) {
      wait_for_event(&event);      /* jedyna możliwość to frame_arrival */
      from_physical_layer(&r);     /* idź pobrać przychodzącą ramkę */
      to_network_layer(&r.info);   /* przekaż dane do warstwy sieciowej */
      to_physical_layer(&s);       /* wyślij pustą ramkę, żeby obudzić nadajnik */
   }
}
```

Wprawdzie dane w tym przykładzie są transmitowane w jednym kierunku, z nadajnika do odbiornika, lecz ramki podróżują w obie strony. Wobec tego kanał komunikacyjny pomiędzy dwiema warstwami łącza danych musi być zdolny do przesyłu informacji w obie strony. Jednakże ten protokół wymaga ścisłej przemienności przepływu — najpierw nadajnik wysyła ramkę, następnie odbiornik wysyła ramkę, potem nadajnik, znów odbiornik i tak dalej. Tutaj wystarczy półdupleksowy kanał fizyczny.

Tak jak w protokole 1. wszystko zaczyna się od pobrania przez nadajnik pakietu z warstwy sieciowej. Następnie nadajnik tworzy z tego pakietu ramkę i wysyła ją. Lecz teraz, w przeciwieństwie do protokołu 1., nadajnik musi czekać na nadejście ramki potwierdzającej, zanim wróci na początek pętli i pobierze następną ramkę z warstwy sieciowej. Warstwa łącza danych nadajnika nie musi nawet analizować otrzymanej ramki — możliwość jest tylko jedna. Przychodząca ramka jest zawsze potwierdzeniem.

Jedyna różnica pomiędzy `receiver1` i `receiver2` polega na tym, że po doręczeniu pakietu do warstwy sieciowej `receiver2` odsyła do nadajnika ramkę potwierdzającą, zanim ponownie wejdzie do pętli oczekującej. Ponieważ ważne jest jedynie nadejście ramki z powrotem do nadajnika, a nie jej treść, odbiornik nie musi umieszczać w niej żadnych konkretnych informacji.

3.3.3. Protokół simpleksowy dla kanału z zakłóceniami

Rozważmy teraz typową sytuację, w której kanał komunikacyjny wprowadza błędy. Ramki mogą być albo uszkodzone, albo całkowicie tracone. Jednakże założymy, że jeśli ramka uległa uszkodzeniu podczas transmisji, sprzęt odbiornika wykryje to po obliczeniu sumy kontrolnej. Jeśli ramka jest uszkodzona w taki sposób, że suma kontrolna będzie mimo to poprawna (mało prawdopodobne), to protokół ten, podobnie jak każdy inny, może zawieść (tzn. przekazać uszkodzony pakiet do warstwy sieciowej).

Na pierwszy rzut oka może wydawać się, że wystarczy nam modyfikacja protokołu 2. przez dodanie czasomierza. Nadajnik może wysłać ramkę, lecz odbiornik wyśle potwierdzenie tylko wtedy, gdy otrzyma poprawne dane. Jeśli do odbiornika dotrze uszkodzona ramka, to zostanie odrzucona. Po jakimś czasie upłynie okres oczekiwania i nadajnik wyśle ramkę jeszcze raz. Taki proces byłby powtarzany aż do momentu, gdy ramka w końcu dotrze nieuszkodzona.

Powyższy schemat zawiera poważny błąd. Proszę pomyśleć i spróbować znaleźć problem przed kontynuowaniem lektury.

Aby zorientować się, co może pójść nie tak, przypomnijmy sobie, że zadaniem warstwy łącza danych jest zapewnienie wolnej od błędów i przezroczystej komunikacji pomiędzy procesami warstwy sieciowej. Warstwa sieciowa w komputerze *A* przekazuje serię pakietów do swojej warstwy łącza danych, która musi zagwarantować doręczenie identycznej serii pakietów do warstwy sieciowej komputera *B* poprzez jego warstwę łącza danych. Przede wszystkim warstwa sieciowa w *B* nie ma sposobu na rozpoznanie, czy pakiet nie został utracony lub powielony, więc warstwa łącza danych musi zagwarantować, że żadna kombinacja błędów transmisji, nawet najmniej prawdopodobna, nie spowoduje doręczenia duplikatu pakietu do warstwy sieciowej.

Rozważmy taki scenariusz:

(1) Warstwa sieciowa w *A* przekazuje pakiet nr 1 do swojej warstwy łącza danych. Pakiet zostaje poprawnie odebrany w *B* i przekazany do warstwy sieciowej w *B*. Komputer *B* odsyła ramkę potwierdzającą do *A*.
(2) Ramka potwierdzająca zostaje całkowicie utracona — w ogóle nie dociera do *A*. Życie byłoby o wiele prostsze, gdyby kanał przekłamywał i tracił tylko ramki danych, a nie sterujące, lecz musimy z przykrością stwierdzić, że kanał nie jest specjalnie wybiórczy pod tym względem.
(3) W warstwie łącza danych w *A* w końcu upływa czas oczekiwania. Ponieważ potwierdzenie nie dotarło, warstwa zakłada (błędnie), że jej ramka danych została utracona lub uszkodzona i wysyła ponownie ramkę zawierającą pakiet nr 1.
(4) Powielona ramka również dociera do warstwy łącza danych w *B* bez problemów i zostaje nieświadomie przekazana do warstwy sieciowej. Jeśli *A* wysyła plik do *B*, część pliku będzie zdublowana, czyli kopia pliku utworzona w *B* będzie błędna, a błąd ten nie zostanie wykryty. Inaczej mówiąc, protokół zawiedzie.

Najwyraźniej potrzeba tu jakiejś metody pozwalającej odbiornikowi odróżnić ramkę, którą otrzymuje po raz pierwszy, od retransmisji. Oczywistym sposobem na to jest umieszczenie przez nadajnik numeru sekwencyjnego w każdej wysyłanej przez siebie ramce. Odbiornik będzie mógł skontrolować numer każdej otrzymywanej ramki, aby sprawdzić, czy jest to nowa ramka czy duplikat, który powinien odrzucić.

Ponieważ pożądany jest jak najmniejszy rozmiar nagłówka, powstaje pytanie: jaka minimalna liczba bitów jest potrzebna na numer sekwencyjny? Jedyna niejednoznaczność w naszym protokole występuje pomiędzy ramką (m) i jej następcą ($m + 1$). Jeśli ramka m zostanie utracona lub uszkodzona, odbiornik nie potwierdzi jej, więc nadajnik będzie ją nadawał jeszcze raz. Po otrzymaniu poprawnej ramki odbiornik wyśle potwierdzenie do nadajnika. Tutaj pojawia się potencjalny problem. W zależności od tego, czy ramka potwierdzająca dotrze do nadajnika czy nie, nadajnik może próbować nadać ramkę m lub $m + 1$.

Zdarzeniem, które wyzwala wysłanie przez nadajnik ramki $m + 2$, jest nadejście potwierdzenia dla ramki $m + 1$. Lecz to implikuje, że ramka m została poprawnie odebrana, a poza tym potwierdzenie również dotarło do nadajnika (w przeciwnym razie nadajnik nie zacząłby nadawać $m + 1$, o $m + 2$ już nie wspominając). W konsekwencji jedyna niejasność występuje pomiędzy ramką i jej bezpośrednim poprzednikiem lub następcą, a nie pomiędzy poprzednikiem i następcą.

Wystarczy więc jednobitowy numer sekwencyjny (0 lub 1). W każdej chwili odbiornik spodziewa się otrzymania konkretnego numeru sekwencyjnego. Każda otrzymana ramka z błędnym

numerem sekwencyjnym jest odrzucana jako duplikat. Gdy przychodzi ramka zawierająca poprawny numer sekwencyjny, zostaje zaakceptowana i przekazana do warstwy sieciowej. Następnie spodziewany numer sekwencyjny jest inkrementowany modulo 2 (tzn. 0 staje się 1, a 1 staje się 0).

Przykład protokołu tego typu przedstawia listing 3.4. Protokoły, w których nadajnik czeka na pozytywne potwierdzenie przed przejściem do następnej jednostki danych, często nazywane są **PAR** (**Positive Acknowledgement with Retransmission** — potwierdzenia pozytywne z retransmisją) lub **ARQ** (**Automatic Repeat reQuest** — automatyczne żądanie ponownego przesłania). Podobnie jak protokół 2., ten również przesyła dane tylko w jednym kierunku.

LISTING 3.4.
Protokół potwierdzeń pozytywnych z retransmisją

```
/* Protokół 3. (par) umożliwia jednokierunkowy przepływ danych zawodnym kanałem. */
#define MAX_SEQ 1                       /* musi być 1 dla protokołu 3. */
typedef enum  {frame_arrival, cksum_err, timeout} event_type;
#include "protocol.h"
void sender3(void)
{
   seq_nr next_frame_to_send;           /* nr sekwencyjny następnej wychodzącej ramki */
   frame s;                             /* zamazywane dane ramki */
   packet buffer;                       /* bufor dla wychodzącego pakietu */
   event_type event;

   next_frame_to_send = 0;              /* inicjuj wychodzące numery sekwencyjne */
   from_network_layer(&buffer);         /* pobierz pierwszy pakiet */
   while (true) {
      s.info = buffer;                  /* zbuduj ramkę do transmisji */
      s.seq = next_frame_to_send;       /* wstaw nr sekwencyjny do ramki */
      to_physical_layer(&s);            /* wyślij w drogę */
      start_timer(s.seq);               /* gdy odpowiedź za długo nie dociera, timeout */
      wait_for_event(&event);           /* frame_arrival, cksum_err, timeout */
      if (event == frame_arrival) {
         from_physical_layer(&s);       /* pobierz potwierdzenie */
         if (s.ack == next_frame_to_send) {
            stop_timer(s.ack);          /* wyłącz czasomierz*/
            from_network_layer(&buffer);   /* pobierz następny do wysłania */
            inc(next_frame_to_send);    /* zaneguj next_frame_to_send */
         }
      }
   }
}

void receiver3(void)
{
   seq_nr frame_expected;
   frame r, s;
   event_type event;

   frame_expected = 0;
   while (true) {
      wait_for_event(&event);           /* możliwości: frame_arrival, cksum_err */
```

```
      if (event == frame_arrival) {   /* dotarła poprawna ramka */
         from_physical_layer(&r);     /* idź pobrać odebraną właśnie ramkę */
         if (r.seq == frame_expected) {   /* na to czekaliśmy */
            to_network_layer(&r.info);    /* przekaż dane do warstwy sieciowej */
            inc(frame_expected);      /* następnym razem oczekuj drugiego numeru sek. */
         }
         s.ack = 1 - frame_expected;  /* powiedz, która ramka jest potwierdzana */
         to_physical_layer(&s);       /* wyślij potwierdzenie */
      }
   }
}
```

Protokół 3. różni się od swoich poprzedników tym, że zarówno nadajnik, jak i odbiornik mają zmienną, której wartość jest pamiętana w stanie oczekiwania warstwy łącza danych. Nadajnik pamięta numer sekwencyjny następnej ramki, którą ma wysłać, w `next_frame_to_send`; odbiornik pamięta numer sekwencyjny oczekiwanej ramki w `frame_expected`. Każdy protokół ma krótką fazę inicjalizacji przed wejściem do pętli nieskończonej.

Po wysłaniu ramki nadajnik uruchamia czasomierz. Jeśli czasomierz już działał, to zostaje wyzerowany, aby odmierzył kolejny pełny interwał. Interwał czasu powinien być tak dobrany, by wystarczył na dotarcie ramki do odbiornika, przetworzenie tej ramki przez odbiornik w najbardziej niekorzystnych warunkach i propagację ramki potwierdzającej z powrotem do nadawcy. Dopiero po upłynięciu tego czasu nadajnik może bezpiecznie założyć, że albo ramka danych, albo potwierdzająca została utracona, i wysłać duplikat. Jeśli interwał czasomierza będzie zbyt krótki, nadajnik będzie wysyłał niepotrzebne ramki. Wprawdzie nie wpłyną one na poprawność protokołu, lecz zaszkodzą wydajności.

Po wysłaniu ramki i uruchomieniu czasomierza nadajnik czeka na jakieś ekscytujące zdarzenie. Istnieją tylko trzy możliwości: przyjdzie nieuszkodzona ramka potwierdzająca, doczołga się uszkodzona ramka potwierdzająca albo upłynie czas oczekiwania. Jeśli dotrze poprawne potwierdzenie, nadajnik pobierze następny pakiet z warstwy sieciowej i umieści go w buforze, zamazując poprzedni pakiet, oraz zwiększy numer sekwencyjny. Jeśli dotrze uszkodzona ramka lub nie dotrze żadna, ani zawartość bufora, ani numer sekwencyjny nie zostaną zmienione, co pozwoli wysłać duplikat.

Gdy do odbiornika dociera poprawna ramka, jej numer sekwencyjny jest sprawdzany, aby upewnić się, czy ramka nie jest duplikatem. Jeśli nie, to zostaje zaakceptowana, przekazana do warstwy sieciowej i generowane jest potwierdzenie. Duplikaty i ramki uszkodzone nie są przekazywane do warstwy sieciowej.

3.4. Protokoły z oknem przesuwnym

W poprzednich protokołach ramki danych były przesyłane tylko w jednym kierunku. W większości sytuacji istnieje potrzeba transmisji danych w obie strony. Jednym ze sposobów na otrzymanie transmisji pełnodupleksowej jest utworzenie dwóch odrębnych kanałów komunikacyjnych i używanie każdego na potrzeby jednokierunkowej transmisji danych (w przeciwnych kierunkach). W takiej sytuacji mamy dwa odrębne obwody fizyczne, każdy z kanałem „docelowym" (dla danych) i „zwrotnym" (na potwierdzenia). W obu przypadkach pasmo kanału zwrotnego marnuje się niemal w całości. Skutek jest taki, że użytkownik płaci za dwa obwody, lecz używa możliwości jednego.

Lepszym pomysłem będzie użycie tego samego obwodu do transmisji danych w obie strony. W końcu w protokołach 2. i 3. jest on już używany do przesyłania ramek tam i z powrotem, a kanał zwrotny ma tę samą przepustowość co kanał docelowy. W tym modelu ramki danych z *A* do *B* są przemieszane z ramkami potwierdzającymi z *A* dla *B*. Sprawdzając pole `kind` w nagłówku ramki przychodzącej, odbiornik może odróżnić ramkę danych od potwierdzającej.

Wprawdzie przeplatanie ramek danych i ramek kontrolnych w tym samym obwodzie jest postępem w stosunku do konieczności używania dwóch odrębnych obwodów fizycznych, lecz możliwe jest jeszcze dalsze udoskonalenie. Gdy przychodzi ramka danych, zamiast natychmiast odesłać odrębną ramkę potwierdzającą, nadajnik powstrzymuje się od tego i czeka na kolejny pakiet otrzymany z warstwy sieciowej. Potwierdzenie zostaje dołączone do wychodzącej ramki danych (poprzez użycie pola `ack` w nagłówku ramki). W wyniku potwierdzenie wiezie się za darmo w następnej wychodzącej ramce danych. Ta technika czasowego opóźnienia wychodzących potwierdzeń tak, by mogły zostać podpięte do następnej wychodzącej ramki danych, nosi nazwę **piggybacking** („jazda na barana").

Podstawową zaletą tej metody względem osobnych ramek potwierdzających jest lepsze wykorzystanie dostępnego pasma kanału. Pole `ack` w nagłówku ramki kosztuje tylko kilka bitów, podczas gdy osobna ramka wymagałaby nagłówka, potwierdzenia i sumy kontrolnej. Oprócz tego mniejsza liczba wysłanych ramek oznacza mniej przerwań „przybycie ramki", i być może mniej buforów w odbiorniku, zależnie od organizacji oprogramowania odbiornika. W następnym protokole, którym się zajmiemy, metoda „na barana" będzie kosztować tylko 1 bit w nagłówku ramki. Rzadko wymaga więcej niż kilku bitów.

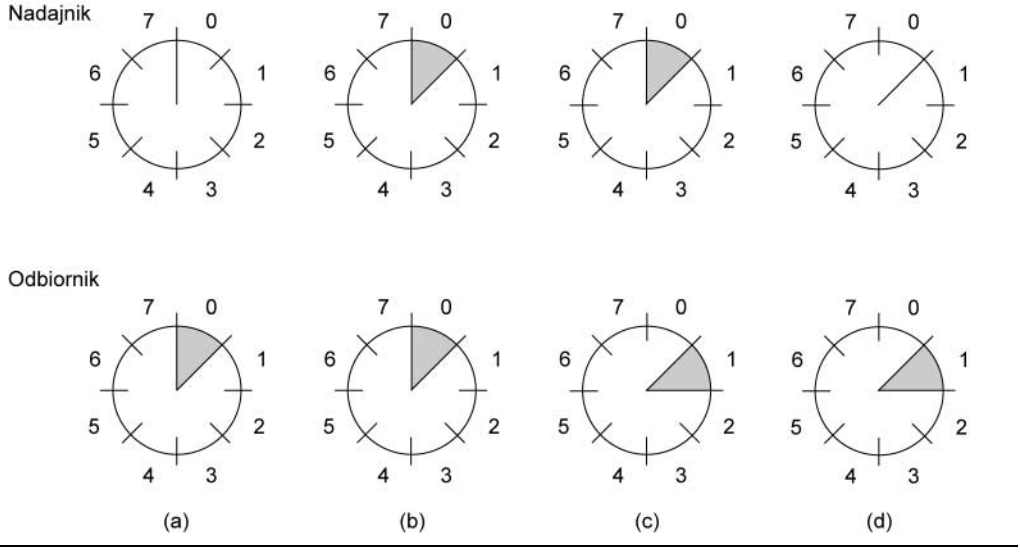
Jednakże technika ta wprowadza komplikację niewystępującą przy odrębnych potwierdzeniach. Jak długo warstwa łącza danych powinna czekać na pakiet, na którym może „przewieźć" potwierdzenie? Jeśli warstwa łącza danych będzie czekać dłużej niż wynosi okres oczekiwania nadajnika, ramka zostanie nadana ponownie, negując całą ideę korzystania z potwierdzeń. Gdyby warstwa łącza danych była wyrocznią i mogła przepowiadać przyszłość, to wiedziałaby, kiedy pojawi się następny pakiet z warstwy sieciowej, co pozwoliłoby zdecydować, czy czekać na niego, czy też wysłać osobne potwierdzenie, zależnie od spodziewanego czasu oczekiwania. Oczywiście warstwa łącza danych nie potrafi przepowiadać przyszłości, więc musi skorzystać z jakiegoś doraźnego schematu, na przykład czekać ustaloną liczbę milisekund. Jeśli nowy pakiet pojawi się szybko, potwierdzenie zostanie przewiezione na nim; w przeciwnym razie, jeśli do końca ustalonego czasu oczekiwania nie pojawi się żaden nowy pakiet, warstwa łącza danych po prostu wyśle odrębną ramkę potwierdzającą.

Następne trzy protokoły są dwukierunkowe i należą do klasy tzw. **protokołów z oknem przesuwnym** (ang. *sliding window*). Te trzy przykłady różnią się wydajnością, złożonością i zapotrzebowaniem na bufory, co omówimy później. W nich, podobnie jak we wszystkich protokołach z oknem przesuwnym, każda wysyłana ramka zawiera numer sekwencyjny mieszczący się w zakresie od 0 do jakiegoś maksimum. Maksimum zwykle wynosi $2^n - 1$, więc numer sekwencyjny mieści się dokładnie w n-bitowym polu. Protokół z oknem przesuwnym typu stop-and-wait używa $n = 1$, co ogranicza numery sekwencyjne do 0 i 1, lecz bardziej wyrafinowane wersje mogą używać dowolnych wartości n.

Kwintesencją wszelkich protokołów z oknem przesuwnym jest to, że w każdej chwili nadajnik pamięta zbiór numerów sekwencyjnych odpowiadających ramkom, które ma prawo wysłać. Mówimy, że ramki te mieszczą się w **oknie nadawczym**. Podobnie odbiornik utrzymuje **okno odbiorcze** odpowiadające zbiorowi ramek, które ma prawo przyjąć. Okna nadajnika i odbiornika nie muszą mieć tych samych granic górnych i dolnych, a nawet takiej samej wielkości. W niektórych protokołach mają stały rozmiar, lecz w innych mogą z czasem zwiększać lub zmniejszać wielkość w miarę wysyłania i odbierania ramek.

Wprawdzie protokoły te dają warstwie łącza danych większą swobodę kolejności, w której może odbierać i nadawać ramki, lecz zdecydowanie nie zarzuciliśmy wymogu, że protokół musi doręczyć pakiety do docelowej warstwy sieciowej w tej samej kolejności, w jakiej zostały przekazane do warstwy łącza danych w komputerze nadającym dane. Zmieniliśmy teraz wymóg „liniowości" kanału komunikacyjnego, to znaczy tego, że musi doręczyć wszystkie ramki w kolejności wysłania.

Numery sekwencyjne w obrębie okna nadawczego reprezentują ramki, które zostały wysłane lub mogą zostać wysłane, lecz nie zostały jeszcze potwierdzone. Za każdym razem, gdy nowy pakiet przychodzi z warstwy sieciowej, otrzymuje najwyższy kolejny numer sekwencyjny, a górna krawędź okna jest zwiększana o jeden. Gdy przychodzi potwierdzenie, dolna krawędź jest zwiększana o jeden. W ten sposób okno nieustannie utrzymuje listę niepotwierdzonych ramek. Przykład przedstawia rysunek 3.9.



RYSUNEK 3.9. Okno przesuwne o rozmiarze 1 z 3-bitowym numerem sekwencyjnym: (a) stan początkowy, (b) po wysłaniu pierwszej ramki, (c) po odebraniu pierwszej ramki, (d) po odebraniu potwierdzenia pierwszej ramki

Ponieważ ramki znajdujące się aktualnie w oknie nadajnika mogą zostać utracone lub uszkodzone podczas transmisji, nadajnik musi przechowywać wszystkie te ramki w pamięci na wypadek potrzeby retransmisji. Wobec tego, jeśli maksymalna wielkość okna wynosi *n*, nadajnik potrzebuje *n* buforów do przechowania niepotwierdzonych ramek. Jeśli okno kiedykolwiek zwiększy swój rozmiar do maksymalnego, warstwa łącza danych nadajnika musi siłą zablokować warstwę sieciową, dopóki nie zwolni się kolejny bufor.

Okno warstwy łącza danych odbiornika odpowiada ramkom, które może przyjąć. Każda ramka spoza okna jest odrzucana bez komentarza. Gdy zostanie odebrana ramka, której numer sekwencyjny jest równy dolnej krawędzi okna przesuwnego, ramka zostaje przekazana do warstwy sieciowej, zostaje wygenerowane potwierdzenie i okno jest przekręcane o jeden. Inaczej niż w oknie nadajnika okno odbiornika zawsze zachowuje początkową wielkość. Proszę zwrócić uwagę, że okno o rozmiarze 1 oznacza, iż warstwa łącza danych przyjmuje ramki tylko po kolei, lecz dla większych okien tak nie jest. Z kolei warstwa sieciowa zawsze otrzymuje dane we właściwym porządku, niezależnie od rozmiaru okna warstwy łącza danych.

Rysunek 3.9 pokazuje przykład z maksymalnym rozmiarem okna równym 1. Na początku nie oczekują żadne ramki, więc górna i dolna krawędź okna są sobie równe, lecz z upływem czasu sytuacja zmienia się jak na rysunku.

3.4.1. Protokół z jednobitowym oknem przesuwnym

Zanim zabierzemy się za przypadek ogólny, przeanalizujmy najpierw protokół z oknem przesuwnym o maksymalnym rozmiarze równym 1. Jest to protokół typu stop-and-wait, ponieważ nadajnik wysyła ramkę i czeka na potwierdzenie przed wysłaniem kolejnej.

Protokół taki przestawia listing 3.5. Podobnie jak poprzednie zaczyna się od zdefiniowania zmiennych. `next_frame_to_send` mówi, którą ramkę nadajnik usiłuje wysłać. Analogicznie, `frame_expected` informuje, której ramki spodziewa się odbiornik. W obu przypadkach możliwe są tylko 1 lub 0.

LISTING 3.5.
Protokół z jednobitowym oknem przesuwnym

```
/* Protokół 4. (z oknem przesuwnym) jest dwukierunkowy. */
#define MAX_SEQ 1                       /* musi być 1 dla protokołu 4. */
typedef enum {frame_arrival, cksum_err, timeout} event_type;
#include "protocol.h"
void protocol4 (void)
{
   seq_nr next_frame_to_send;           /* tylko 0 lub 1 */
   seq_nr frame_expected;               /* tylko 0 lub 1 */
   frame r, s;                          /* zmienne tymczasowe */
   packet buffer;                       /* bieżący pakiet jest wysyłany */
   event_type event;
   next_frame_to_send = 0;              /* następna ramka w strumieniu wyjściowym */
   frame_expected = 0;                  /* następna oczekiwana ramka */
   from_network_layer(&buffer);         /* pobierz pakiet z warstwy sieciowej */
   s.info = buffer;                     /* przygotuj się do wysłania pierwszej ramki */
   s.seq = next_frame_to_send;          /* wstaw numer sekwencyjny do ramki */
   s.ack = 1 - frame_expected;          /* ack na barana */
   to_physical_layer(&s);               /* nadaj ramkę */
   start_timer(s.seq);                  /* uruchom czasomierz */
   while (true) {
      wait_for_event(&event);           /* frame_arrival, cksum_err lub timeout */
      if (event == frame_arrival) {     /* ramka dotarła bez uszkodzeń. */
         from_physical_layer(&r);           /* pobierz ją */
         if (r.seq == frame_expected) {     /* obsłuż wejściowy strumień ramek. */
           to_network_layer(&r.info);    /* przekaż pakiet do warstwy sieciowej */
           inc(frame_expected);          /* zaneguj następny oczekiwany nr sekwencyjny */
      }
      if (r.ack == next_frame_to_send) {    /* obsłuż wyjściowy strumień ramek. */
         stop_timer(r.ack);             /* wyłącz czasomierz */
         from_network_layer(&buffer); /* pobierz nowy pakiet z warstwy sieciowej */
         inc(next_frame_to_send);     /* zaneguj nr sekwencyjny nadajnika */
         }
      }
      s.info = buffer;                  /* zbuduj wychodzącą ramkę */
      s.seq = next_frame_to_send;       /* wstaw do niej nr sekwencyjny */
      s.ack = 1 - frame_expected;       /* nr sekwencyjny ostatniej odebranej ramki */
      to_physical_layer(&s);            /* wyślij ramkę */
      start_timer(s.seq);               /* uruchom czasomierz */
   }
}
```

W normalnych warunkach jedna z dwóch warstw łącza danych jako pierwsza wysyła pierwszą ramkę. Inaczej mówiąc, tylko jeden z dwóch programów warstwy łącza danych powinien zawierać wywołania procedur `to_physical_layer` i `start_timer` na zewnątrz pętli głównej. W przypadku gdy obie warstwy łącza danych startują jednocześnie, pojawia się osobliwa sytuacja, którą omówimy później. Komputer rozpoczynający pobiera pierwszy pakiet z warstwy sieciowej, buduje z niego ramkę i wysyła ją. Gdy ta (lub dowolna) ramka dotrze, warstwa łącza danych odbiornika sprawdza, czy nie jest duplikatem, tak jak w protokole 3. Jeśli jest to spodziewana ramka, zostaje ona przekazana do warstwy sieciowej i okno odbiornika przesuwa się w górę.

Pole potwierdzenia zawiera numer ostatniej bezbłędnie odebranej ramki Jeśli ten numer zgadza się z numerem ramki, którą nadajnik próbuje wysłać, to dla nadajnika oznacza to, że skończył przetwarzać ramkę zapisaną w zmiennej `buffer` i może pobrać kolejną ramkę ze swojej warstwy sieciowej. Jeśli numer sekwencyjny jest niezgodny, nadajnik musi ponawiać próbę wysłania tej samej ramki. Zawsze, gdy zostaje odebrana ramka, jest również odsyłana inna ramka.

Przeanalizujmy teraz protokół 4., aby zobaczyć, jak jest odporny na patologiczne scenariusze. Załóżmy, że komputer *A* chce wysłać swoją ramkę 0 do komputera *B*, a *B* próbuje wysłać ramkę 0 do *A*. Załóżmy, że *A* wysyła ramkę do *B*, lecz interwał oczekiwania *A* jest trochę za krótki. Wobec tego w *A* może raz za razem upływać czas oczekiwania, przez co *A* będzie wysyłać serię identycznych ramek z `seq` = 0 i `ack` = 1.

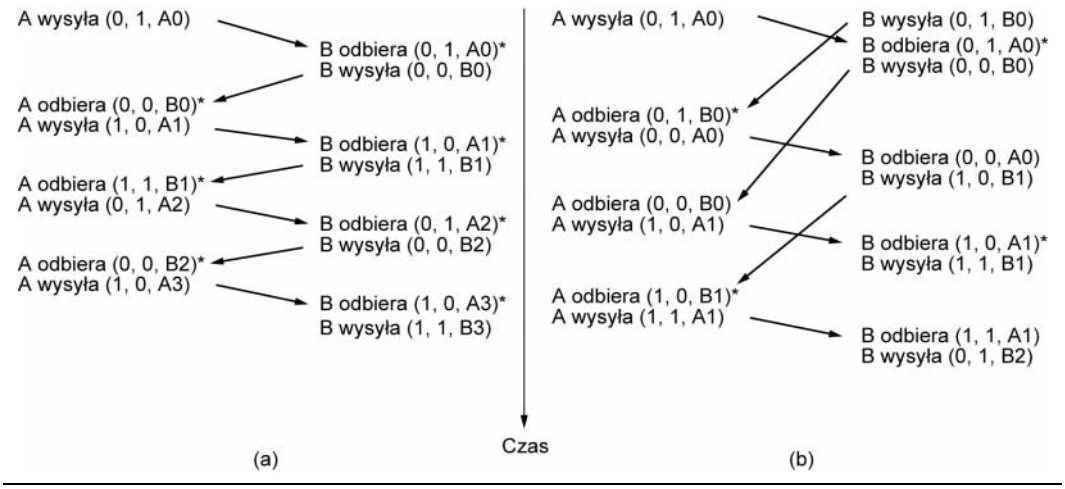
Gdy pierwsza poprawna ramka dotrze do komputera *B*, zostanie przyjęta i `frame_expected` przyjmie wartość 1. Wszystkie kolejne ramki zostaną odrzucone, ponieważ *B* spodziewa się teraz ramki o numerze sekwencyjnym 1, a nie 0. Co więcej, ponieważ wszystkie duplikaty będą miały `ack` = 1, a *B* nadal czeka na potwierdzenie 0, *B* nie pobierze nowego pakietu ze swojej warstwy sieciowej.

Po przyjściu każdego odrzuconego duplikatu *B* wysyła do *A* ramkę zawierającą `seq` = 0 i `ack` = 0. W końcu jedna z nich dotrze poprawnie do *A*, co spowoduje, że *A* zacznie nadawać kolejną ramkę. Żadna kombinacja utraconych ramek i zbyt krótkich limitów czasowych nie może spowodować, że protokół doręczy dwukrotnie pakiet do dowolnej warstwy sieciowej, pominie pakiet lub się zakleszczy.

Jednakże pojawia się osobliwa sytuacja, gdy obie strony jednocześnie wysyłają pierwszy pakiet. Ten problem z synchronizacją przedstawia rysunek 3.10. Część (a) pokazuje normalne działanie protokołu. Część (b) ilustruje wspomnianą osobliwość. Jeśli *B* czeka na pierwszą ramkę *A* przed wysłaniem własnej, to sekwencja wygląda jak w (a) i wszystkie ramki zostają zaakceptowane. Jeśli jednak *A* i *B* jednocześnie zainicjują komunikację, ich pierwsze ramki dotrą na miejsce, a następnie warstwy łącza danych dojdą do sytuacji (b). W (a) każde nadejście ramki przynosi nowy pakiet dla warstwy sieciowej i nie ma duplikatów. W (b) połowa ramek zawiera duplikaty mimo to, że nie występują błędy transmisji. Podobna sytuacja może wystąpić na skutek zbyt krótkich limitów czasowych, nawet gdy jedna strona ewidentnie zacznie jako pierwsza. W rzeczy samej kilkakrotne wystąpienie zbyt krótkich limitów czasowych może spowodować wysłanie ramki trzy lub więcej razy.

3.4.2. Protokół używający techniki „wróć do n”

Do tej pory przyjmowaliśmy milczące założenie, że czas transmisji potrzebny na dotarcie ramki do odbiornika razem z czasem transmisji z powrotem potwierdzenia jest pomijalny. Czasami takie założenie jest wyraźnie błędne. W takich sytuacjach długi czas podróży w dwie strony może mieć poważny wpływ na skuteczność wykorzystania pasma. Weźmy na przykład kanał satelitarny 50 kb/s z opóźnieniem propagacji tam i z powrotem równym 500 ms. Załóżmy, że spróbujemy użyć protokołu 4. do przesyłania satelitą 1000-bitowych ramek. W chwili $t = 0$ nadajnik zaczyna wysyłać pierwszą ramkę. W chwili $t = 20$ ms kończy się wysyłanie całej ramki. Dopiero w $t = 270$ ms cała ramka dociera do odbiornika, a dopiero w chwili $t = 520$ ms potwierdzenie wraca do nadajnika (w najlepszych warunkach — bez czekania w odbiorniku i przy krótkiej ramce potwierdzającej). Oznacza



RYSUNEK 3.10. Dwa scenariusze dla protokołu 4.: (a) standardowy, (b) nietypowy. W nawiasach zapisane są kolejno (seq, ack, nr pakietu). Gwiazdka oznacza miejsce przyjęcia pakietu przez warstwę sieciową

to, że nadajnik był zablokowany przez 500/520 = 96% czasu. Inaczej mówiąc, tylko 4 procent dostępnego pasma zostało wykorzystane. Widać wyraźnie, że połączenie długiego czasu przejścia, wysokiej przepustowości i małej długości ramek jest katastrofalne, gdy idzie o wydajność.

Opisany powyżej problem możemy uznać za konsekwencję reguły, która żąda od nadajnika czekania na ramkę potwierdzającą przed wysłaniem kolejnej. Jeśli złagodzimy to ograniczenie, będzie można osiągnąć znacznie wyższą wydajność. Rozwiązanie polega zasadniczo na zezwoleniu nadajnikowi na wysłanie maksymalnie w ramek przed zablokowaniem transmisji zamiast tylko jednej ramki. Odpowiednio dobrana wartość w pozwoli nadajnikowi wysyłać ramki jedna za drugą przez czas równy czasowi przejścia w obie strony bez zapełnienia okna. W powyższym przykładzie wartość w powinna wynosić przynajmniej 26. Nadajnik zaczyna od wysłania ramki 0, jak powyżej. Do chwili zakończenia wysyłania 26 ramek, w chwili t = 520, powinna właśnie wrócić ramka potwierdzająca dla ramki 0. Następne potwierdzenia będą docierać co 20 ms, więc nadajnik zawsze będzie miał pozwolenie na wysłanie kolejnej ramki, gdy będzie tego potrzebował. Przez cały czas 25 lub 26 ramek będzie oczekiwać na potwierdzenie. Inaczej mówiąc, maksymalny rozmiar okna nadajnika wynosi 26.

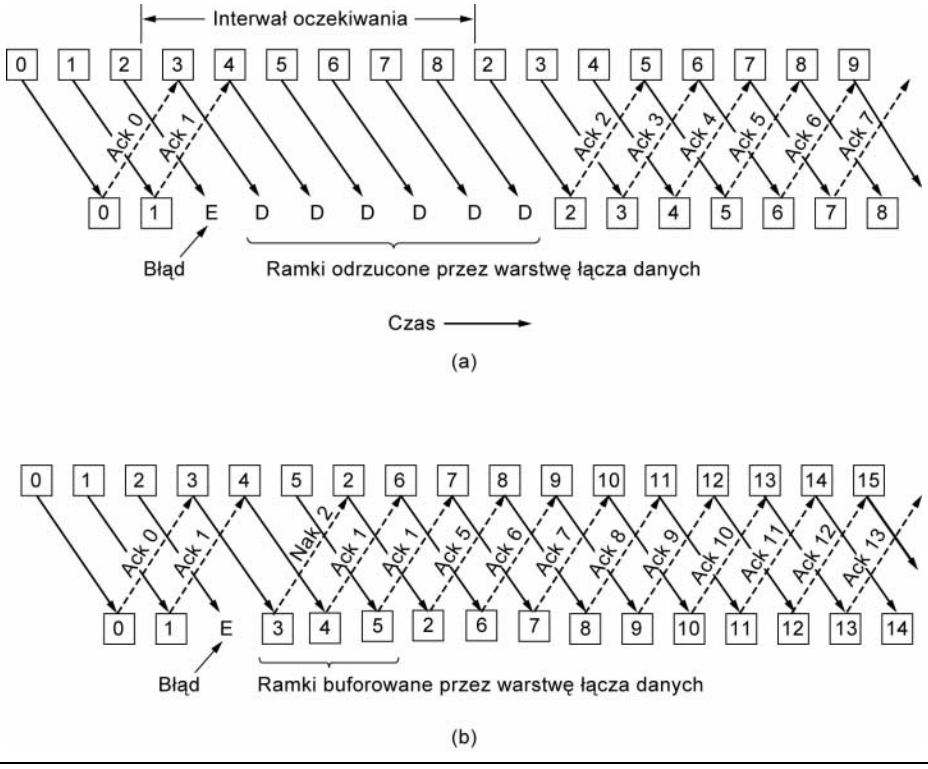
Potrzeba większego okna po stronie nadającej występuje zawsze, gdy iloczyn pasma i czasu podróży w obie strony jest wysoki. Gdy pasmo jest wysokie, nawet przy umiarkowanych opóźnieniach nadajnik wyczerpie swoje okno dość szybko, chyba że dysponuje dużym oknem. Jeśli opóźnienie jest wysokie (jak w kanałach satelitów geostacjonarnych), nadajnik wyczerpie swoje okno nawet przy umiarkowanym paśmie. Iloczyn tych dwóch elementów zasadniczo decyduje o „objętości rurociągu", a nadajnik musi być w stanie wypełnić „rurę" bez zatrzymywania się, aby pracować z najwyższą wydajnością.

Ta technika nosi nazwę **pracy potokowej** (ang. *pipelining*). Jeśli przepustowość kanału wynosi b bitów na sekundę, wielkość ramki l bitów, a czas propagacji w obie strony R sekund, to czas wymagany na wysłanie jednej ramki wynosi l/b sekund. Po wysłaniu ostatniego bitu ramki danych występuje opóźnienie $R/2$, zanim bit dotrze do odbiornika, oraz opóźnienie równe przynajmniej $R/2$, zanim potwierdzenie wróci do nadajnika, co daje łączne opóźnienie R. W technice stop-and-wait linia jest zajęta przez l/b i bezczynna przez czas R, co daje:

$$\text{wykorzystanie linii} = l/(l + bR)$$

Jeśli $l < bR$, to wydajność będzie niższa od 50% Ponieważ zawsze występuje niezerowe opóźnienie propagacji potwierdzenia, praca potokowa może w zasadzie posłużyć do zajęcia linii w tym okresie, lecz jeśli okres ten jest krótki, dodatkowa złożoność nie jest warta zachodu.

Potokowe przesyłanie ramek zawodnym kanałem komunikacyjnym jest przyczyną pewnych poważnych problemów. Po pierwsze, co stanie się, gdy ramka w środku długiego strumienia zostanie uszkodzona lub utracona? Do odbiornika dotrze duża liczba późniejszych ramek, zanim nadajnik dowie się o problemie. Gdy uszkodzona ramka dociera do odbiornika, powinna oczywiście zostać odrzucona, lecz co powinien zrobić odbiornik z wszystkimi poprawnymi ramkami następującymi po niej? Proszę pamiętać, że warstwa łącza danych w odbiorniku jest zobowiązana doręczać pakiety do warstwy sieciowej we właściwej kolejności. Rysunek 3.11 przedstawia wpływ pracy potokowej na usuwanie błędów. Przyjrzymy się temu bardziej szczegółowo.



RYSUNEK 3.11. Praca potokowa i usuwanie błędów. Wpływ błędu, gdy (a) rozmiar okna odbiornika wynosi 1, oraz (b), gdy rozmiar okna odbiornika jest duży

Dostępne są dwie podstawowe metody radzenia sobie z błędami w pracy potokowej. Jedna z nich, o nazwie **go back n** („wróć do n"), polega na tym, że odbiornik po prostu odrzuca wszystkie kolejne ramki, nie wysyłając potwierdzeń dla odrzuconych ramek. Ta strategia odpowiada oknu odbioru o rozmiarze 1. Inaczej mówiąc, warstwa łącza danych odmawia przyjęcia jakiejkolwiek ramki poza następną, którą musi przekazać do warstwy sieciowej. Jeśli okno nadajnika wypełni się przed upłynięciem dopuszczalnego czasu oczekiwania, potok zacznie się opróżniać. W końcu upłynie czas oczekiwania i nadajnik ponownie wyśle wszystkie niepotwierdzone ramki we właściwej kolejności, zaczynając od ramki uszkodzonej lub utraconej. Takie podejście może zmarnować mnóstwo pasma, jeśli stopa błędów jest wysoka.

Na rysunku 3.11 (a) widzimy przypadek, gdy okno odbiornika jest duże. Ramki 0 i 1 zostały poprawnie odebrane i potwierdzone. Jednakże ramka 2 została uszkodzona lub zgubiona. Nadajnik, nie wiedząc o tym problemie, wysyła ramki dalej, póki nie upłynie czas oczekiwania dla ramki 2. Następnie wraca do ramki 2 i zaczyna od nowa nadawać 2, 3, 4 itd.

Druga ogólna strategia obsługi błędów przy potokowym przesyłaniu ramek nosi nazwę **powtórzeń selektywnych** (ang. *selective repeat*). Gdy używana jest ta technika, odebrana błędna ramka zostaje odrzucona, lecz dobre ramki odebrane po niej są buforowane. Gdy kończy się czas oczekiwania nadajnika, zostaje ponownie przesłana tylko najstarsza niepotwierdzona ramka. Jeśli ta ramka dotrze bez błędów, odbiornik będzie mógł doręczyć do warstwy sieciowej kolejno wszystkie ramki, które buforował. Powtórzenia selektywne są często łączone z koniecznością wysłania przez odbiornik potwierdzenia negatywnego (NAK) po wykryciu błędu, na przykład błędnej sumy kontrolnej lub ramki poza kolejnością. Potwierdzenia negatywne stymulują retransmisje przed upłynięciem czasu oczekiwania, przez co poprawiają wydajność.

Na rysunku 3.11 (b) ramki 0 i 1 również zostały poprawnie odebrane i potwierdzone, a ramka 2 została stracona. Gdy do odbiornika dotarła ramka 3, jego warstwa łącza danych zauważyła brak ramki, więc wysłała NAK dla ramki 2, lecz zapisała ramkę 3 w buforze. Gdy dotarły ramki 4 i 5, te również były buforowane przez warstwę łącza danych, zamiast zostać przekazane do warstwy sieciowej. W końcu NAK ramki 2 dotarło do nadajnika, który natychmiast wysłał ponownie tę ramkę. Gdy ta dotarła do odbiornika, warstwa łącza danych otrzymała ramki 2, 3, 4 oraz 5 i mogła przesłać je wszystkie do warstwy sieciowej we właściwej kolejności. Mogła też potwierdzić wszystkie ramki aż do 5 włącznie, jak na rysunku. Gdyby NAK zostało utracone, w końcu w nadajniku upłynąłby czas oczekiwania na potwierdzenie ramki 2 i nadajnik sam z siebie wysłałby ją (lecz tylko ją), aczkolwiek być może znacznie później. Wobec tego negatywne potwierdzenie przyspieszyło retransmisję tej konkretnie ramki.

Powtórzenia selektywne odpowiadają oknu odbiorczemu większemu niż 1. Każda ramka w obrębie tego okna może zostać przyjęta i buforowana, dopóki wszystkie poprzednie nie zostaną przekazane do warstwy sieciowej. Takie podejście może wymagać sporo pamięci dla warstwy łącza danych, jeśli okno jest duże.

Te dwie alternatywy stanowią kompromisy pomiędzy przepustowością a wielkością buforów warstwy łącza danych. W zależności od tego, którego zasobu bardziej brakuje, może zostać użyta pierwsza lub druga. Listing 3.6 przedstawia protokół potokowy, w którym odbiorcza warstwa łącza danych akceptuje tylko ramki we właściwej kolejności; ramki nadesłane po błędzie są odrzucane. W tym protokole po raz pierwszy zrezygnowaliśmy z założenia, że warstwa sieciowa zawsze ma nieskończoną liczbę pakietów do wysłania. Gdy warstwa sieciowa chce wysłać pakiet, może spowodować wystąpienie zdarzenia `network_layer_ready`. Aby jednak wymusić zasadę sterowania przepływem mówiącą, że w każdej chwili nie może być więcej niż `MAX_SEQ` zaległych ramek, warstwa łącza danych musi być zdolna do powstrzymania warstwy sieciowej przed obciążaniem jej dodatkową pracą. Do tego służą procedury biblioteczne `enable_network_layer` i `disable_network_layer`.

LISTING 3.6.
Protokół z oknem przesuwnym używający techniki „wróć do n”

```
/* Protokół 5. (go back n) pozwala na wiele zalegających ramek. Nadajnik może wysłać
maksymalnie MAX_SEQ ramek bez czekania na potwierdzenie. Oprócz tego, inaczej niż
w poprzednich protokołach, nie zakładamy, że warstwa sieciowa zawsze ma gotowy nowy
pakiet. Zamiast tego warstwa sieciowa powoduje zdarzenie network_layer_ready, gdy ma
pakiet do wysłania. */

#define MAX_SEQ 7                       /* powinna być 2^n - 1 */
typedef enum {frame_arrival, cksum_err, timeout, network_layer_ready} event_type;
```

```
#include "protocol.h"

static boolean between(seq_nr a, seq_nr b, seq_nr c)
{
/* Zwróć true, jeśli a <=b < c cyklicznie; w przeciwnym razie false. */
   if (((a <= b) && (b < c)) || ((c < a) && (a <= b)) || ((b < c) && (c < a)))
      return(true);
   else
   return(false);
}

static void send_data(seq_nr frame_nr, seq_nr frame_expected, packet buffer[])
{
/* Zbuduj i wyślij ramkę danych. */
   frame s;                              /* zmienna tymczasowa */

   s.info = buffer[frame_nr];            /* wstaw pakiet do ramki */
   s.seq = frame_nr;                     /* wstaw nr sekwencyjny do ramki */
   s.ack = (frame_expected + MAX_SEQ) % (MAX_SEQ + 1);   /* potwierdzenie na barana */
   to_physical_layer(&s);                /* wyślij ramkę */
   start_timer(frame_nr);                /* uruchom czasomierz */
}

void protocol5(void)
{
   seq_nr next_frame_to_send;            /* MAX_SEQ > 1; dla strumienia wychodzącego */
   seq_nr ack_expected;                  /* najstarsza jeszcze niepotwierdzona ramka */
   seq_nr frame_expected;               /* nast. ramka oczekiwana w strumieniu wchodzącym */
   frame r;                              /* zmienna tymczasowa */
   packet buffer[MAX_SEQ + 1];           /* bufory dla strumienia wychodzącego */
   seq_nr nbuffered;                     /* liczba aktualnie używanych buforów */
   seq_nr i;                             /* służy do indeksowania w tablicy buforów */
   event_type event;

   enable_network_layer();               /* zezwól na zdarzenia network_layer_ready */
   ack_expected = 0;                     /* następne oczekiwane przychodzące ack */
   next_frame_to_send = 0;               /* następna ramka wychodzi */
   frame_expected = 0;                   /* liczba oczekiwanych ramek */
   nbuffered = 0;                        /* na początku żaden pakiet nie jest buforowany */
   while (true) {
   wait_for_event(&event);               /* cztery możliwości: patrz event_type powyżej */

   switch(event) {
      case network_layer_ready:          /* warstwa sieciowa ma pakiet do wysłania */
         /* Zaakceptuj, zapisz i nadaj nową ramkę. */
         from_network_layer(&buffer[next_frame_to_send]); /* pobierz nowy pakiet */
         nbuffered = nbuffered + 1;      /* rozszerz okno nadajnika */
         send_data(next_frame_to_send, frame_expected, buffer);   /* wyślij ramkę */
         inc(next_frame_to_send);        /* podnieś górną krawędź okna nadajnika */
         break;

      case frame_arrival:                /* nadeszła ramka danych lub sterująca */
         from_physical_layer(&r);        /* weź przychodzącą ramkę z warstwy fizycznej */
```

```
            if (r.seq == frame_expected) {
               /* Ramki są spodziewane tylko po kolei. */
               to_network_layer(&r.info);/* przekaż pakiet do warstwy sieciowej */
               inc(frame_expected);      /* podnieś dolną krawędź okna odbiornika */
            }

            /* Ack n implikuje n - 1, n - 2 itd. Sprawdź to. */
            while (between(ack_expected, r.ack, next_frame_to_send)) {
               /* obsłuż doczepione potwierdzenie */
               nbuffered = nbuffered - 1;/* o jedną ramkę buforowaną mniej */
               stop_timer(ack_expected); /* ramka dotarła OK; zatrzymaj czasomierz */
               inc(ack_expected);        /* Zawęź okno nadajnika */
            }
            break;

         case cksum_err: break;           /* po prostu ignoruj złe ramki */

         case timeout:                   /* problem; retransmituj wszystkie zaległe ramki */
            next_frame_to_send = ack_expected;  /* zacznij retransmisję stąd */
            for (i = 1; i <= nbuffered; i++) {
               send_data(next_frame_to_send, frame_expected, buffer);   /* nadaj ramkę
                                                                        jeszcze raz */
               inc(next_frame_to_send);  /* przygotuj następną do wysłania */
            }
      }
      if (nbuffered < MAX_SEQ)
         enable_network_layer();
      else
         disable_network_layer();
      }
}
```

Proszę zwrócić uwagę, że w każdej chwili może zalegać maksymalnie MAX_SEQ ramek, a nie MAX_SEQ + 1, mimo to, że dostępnych jest MAX_SEQ + 1 różnych numerów sekwencyjnych: 0, 1, 2… MAX_SEQ. Aby zrozumieć, dlaczego takie ograniczenie jest niezbędne, rozważmy poniższy scenariusz z MAX_SEQ = 7.

(1) Nadajnik wysyła ramki od 0 do 7.
(2) Potwierdzenie ramki 7 w końcu dociera „na barana” do nadajnika.
(3) Nadajnik wysyła kolejnych osiem ramek, ponownie z numerami sekwencyjnymi od 0 do 7.
(4) Przychodzi „na barana” kolejne potwierdzenie dla ramki 7.

Pytanie brzmi następująco: Czy wszystkich osiem ramek należących do drugiej porcji dotarło pomyślnie, czy też wszystkich osiem zostało utraconych (licząc ramki odrzucone z powodu błędu jako utracone)? W obu przypadkach odbiornik wysłałby ramkę 7 jako potwierdzenie. Nadajnik nie jest w stanie tego rozróżnić. Z tego powodu maksymalna liczba zaległych ramek musi być ograniczona do MAX_SEQ.

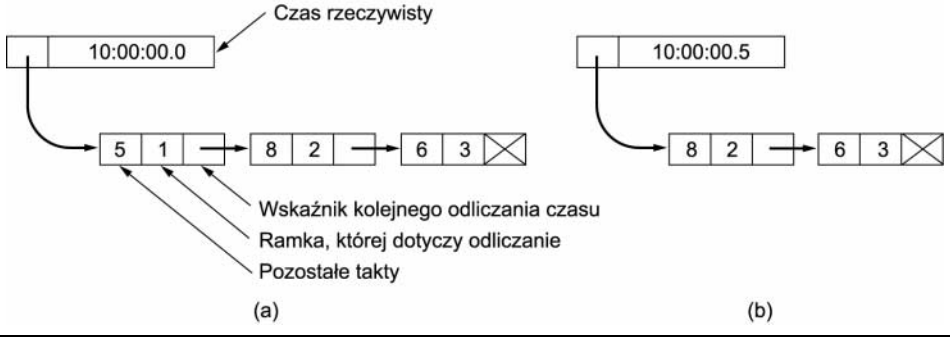
Wprawdzie protokół 5. nie buforuje ramek docierających po błędzie, lecz nie unika całkowicie problemu buforowania. Ponieważ nadajnik może być zmuszony do późniejszej retransmisji wszystkich niepotwierdzonych ramek, to musi przechowywać wszystkie wysłane ramki, dopóki nie uzyska pewności, że zostały przyjęte przez odbiornik. Gdy przychodzi potwierdzenie dla ramki *n*, to ramki

$n-1$, $n-2$ itd. również zostają automatycznie potwierdzone. Ta właściwość jest szczególnie ważna, gdy jakieś poprzednie ramki potwierdzające zostały utracone lub zniekształcone. Zawsze, gdy przychodzi potwierdzenie, warstwa łącza danych sprawdza, czy może teraz zwolnić jakieś bufory. Jeśli tak (tzn. w oknie jest wolne miejsce), uprzednio zablokowana warstwa sieciowa otrzyma zezwolenie na wygenerowanie kolejnych zdarzeń `network_layer_ready`.

Dla tego protokołu zakładamy, że zawsze występuje ruch w drugą stronę, na którym można „podwieźć" potwierdzenia. Jeśli tak nie jest, potwierdzeń nie będzie można wysłać. Protokół 4. nie czyni tego założenia, ponieważ za każdym razem, gdy otrzyma ramkę, nawet uprzednio otrzymaną, odsyła jedną ramkę. W następnym protokole w elegancki sposób rozwiążemy problem z ruchem jednokierunkowym.

Ponieważ w protokole 5. występuje więcej niż jedna zaległa ramka, to logicznie potrzebnych jest więcej czasomierzy, po jednym na zaległą ramkę. Dla każdej ramki czas oczekiwania odliczany jest niezależnie od wszystkich pozostałych. Wszystkie te czasomierze można z łatwością zasymulować programowo z użyciem jednego zegara sprzętowego, który okresowo powoduje przerwania. Odliczane aktualnie czasy oczekiwania tworzą listę powiązaną, której każdy węzeł podaje liczbę taktów zegara do upłynięcia czasu oczekiwania, ramkę, dla której czas jest odliczany i wskaźnik na następny węzeł.

Jako ilustrację tego, jak można zaimplementować te czasomierze, rozważmy przykład z rysunku 3.12 (a). Załóżmy, że zegar odlicza takty co 100 ms. Początkowy czas rzeczywisty to 10:00:00,0; trzy limity czasu kończą się o 10:00:00,5, 10:00:01,3 i 10:00:01,9. Przy każdym takcie zegara sprzętowego aktualizowany jest czas rzeczywisty, a licznik taktów na początku listy jest dekrementowany. Gdy ten licznik wskaże zero, powoduje upłynięcie czasu oczekiwania i węzeł zostaje usunięty z listy, jak na rysunku 3.12 (b). Wprawdzie taka organizacja wymaga skanowania listy przy wywołaniu `start_timer` lub `stop_timer`, lecz nie wymaga zbyt wiele pracy po każdym takcie. W protokole 5. obie te procedury otrzymały parametr wskazujący, dla której ramki ma być mierzony czas.



RYSUNEK 3.12. Symulacja kilku liczników w oprogramowaniu

3.4.3. Protokół używający powtórzeń selektywnych

Protokół 5. sprawdza się, gdy błędy występują rzadko, lecz jeśli linia jest marnej jakości, traci mnóstwo pasma na retransmisje ramek. Alternatywna strategia radzenia sobie z błędami polega na zezwoleniu odbiornikowi na przyjmowanie i buforowanie ramek następujących po ramce uszkodzonej lub zgubionej. Taki protokół nie odrzuca ramek tylko dlatego, że wcześniejsza ramka została uszkodzona lub utracona.

W tym protokole zarówno nadajnik, jak i odbiornik utrzymują okno dopuszczalnych numerów sekwencyjnych. Okno nadajnika zaczyna się od 0 i rośnie do jakiegoś wstępnie zdefiniowanego maksimum, MAX_SEQ. W przeciwieństwie do niego okno odbiornika ma stały rozmiar równy MAX_SEQ. W odbiorniku zarezerwowane są bufory dla wszystkich numerów sekwencyjnych w obrębie tego stałego okna. Z każdym buforem skojarzony jest bit arrived informujący, czy bufor jest pełny czy pusty. Zawsze, gdy przychodzi ramka, jej numer sekwencyjny jest sprawdzany przez funkcję between pod kątem tego, czy mieści się w obrębie okna. Jeśli tak i jeśli nie ramka nie została wcześniej odebrana, to zostaje zaakceptowana i zapamiętana. Ta czynność jest podejmowana niezależnie do tego, czy ramka zawiera kolejny pakiet oczekiwany przez warstwę sieciową czy nie. Oczywiście musi zostać zatrzymana w warstwie łącza danych, a nie przekazana do warstwy sieciowej, dopóki wszystkie poprzedzające ją ramki nie zostaną dostarczone we właściwej kolejności do warstwy sieciowej. Protokół używający tego algorytmu przedstawia listing 3.7.

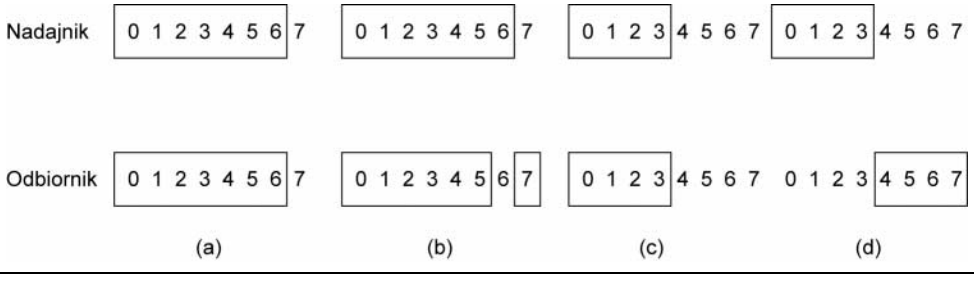
LISTING 3.7.
Protokół z oknem przesuwnym używający powtórzeń selektywnych

```
/* Protokół 6. (z powtórzeniami selektywnymi) przyjmuje ramki poza kolejnością, lecz
przekazuje pakiety do warstwy sieciowej we właściwej kolejności. Z każdą zaległą ramką
skojarzony jest czasomierz. Gdy upłynie czas oczekiwania, ponownie przesyłana jest tylko
dana ramka, a nie wszystkie zaległe, jak w protokole 5. */

#define MAX_SEQ 7                       /* powinno być 2^n - 1 */
#define NR_BUFS ((MAX_SEQ + 1)/2)
typedef enum {frame_arrival, cksum_err, timeout, network_layer_ready, ack_timeout}
event_type;
#include "protocol.h"
boolean no_nak = true;                  /* żadna ramka NAK nie została jeszcze wysłana */
seq_nr oldest_frame = MAX_SEQ + 1;      /* wartość początkowa jest tylko dla symulatora */

static boolean between(seq_nr a, seq_nr b, seq_nr c)
{
/* To samo co between w protokole 5., lecz krótsze i mniej przejrzyste. */
return ((a <= b) && (b < c)) || ((c < a) && (a <= b)) || ((b < c) && (c < a));
}
static void send_frame(frame_kind fk, seq_nr frame_nr, seq_nr frame_expected, packet buffer[])
{
/* Zbuduj i wyślij ramkę danych, ACK lub NAK. */
   frame s;                             /* zmienna tymczasowa */

   s.kind = fk;                         /* kind == data, ack lub nak */
   if (fk == data) s.info = buffer[frame_nr % NR_BUFS];
   s.seq = frame_nr;                    /* ma znaczenie tylko dla ramek danych */
   s.ack = (frame_expected + MAX_SEQ) % (MAX_SEQ + 1);
   if (fk == nak) no_nak = false;       /* proszę tylko jedno NAK na ramkę */
   to_physical_layer(&s);               /* wyślij ramkę */
   if (fk == data) start_timer(frame_nr % NR_BUFS);
   stop_ack_timer();                    /* osobna ramka potwierdzająca niepotrzebna */
}
void protocol6(void)
{
   seq_nr ack_expected;                 /* dolna krawędź okna nadajnika */
   seq_nr next_frame_to_send;           /* górna krawędź okna nadajnika + 1 */
   seq_nr frame_expected;               /* dolna krawędź okna odbiornika */
```

```
seq_nr too_far;                    /* górna krawędź okna odbiornika + 1 */
int i;                             /* indeks do puli buforów */
frame r;                           /* zmienna tymczasowa */
packet out_buf[NR_BUFS];           /* bufory dla strumienia wychodzącego */
packet in_buf[NR_BUFS];            /* bufory dla strumienia wchodzącego */
boolean arrived[NR_BUFS];          /* dane otrzymane */
seq_nr nbuffered;                  /* ile buforów aktualnie w użyciu */
event_type event;

enable_network_layer();            /* inicjalizuj */
ack_expected = 0;  /* następne potwierdzenie spodziewane w strumieniu przychodzącym */
next_frame_to_send = 0;            /* nr następnej ramki wychodzącej */
frame_expected = 0;
too_far = NR_BUFS;
nbuffered = 0;                     /* na początku żaden pakiet nie jest buforowany */
for (i = 0; i < NR_BUFS; i++) arrived[i] = false;
while (true) {
   wait_for_event(&event);         /* pięć możliwości: patrz event_type powyżej*/
   switch(event) {
      case network_layer_ready:    /* zaakceptuj, zapisz i wyślij nową ramkę */
         nbuffered = nbuffered + 1;  /* rozszerz okno */
         from_network_layer(&out_buf[next_frame_to_send % NR_BUFS]); /* pobierz nowy
         pakiet */
         send_frame(data, next_frame_to_send, frame_expected, out_buf);   /* wyślij
         ramkę */
         inc(next_frame_to_send); /* podnieś górną granicę okna */
         break;

      case frame_arrival:          /* dotarła ramka danych lub sterująca */
         from_physical_layer(&r);  /* pobierz ramkę przychodzącą z warstwy fizycznej */
         if (r.kind == data) {
            /* Dotarła nieuszkodzona ramka. */
            if ((r.seq != frame_expected) && no_nak)
               send_frame(nak, 0, frame_expected, out_buf); else start_ack_timer();
            if (between(frame_expected, r.seq, too_far) && (arrived[r.seq%NR_BUFS] ==
            false)) {
               /* Ramki mogą być przyjmowane w dowolnej kolejności. */
               arrived[r.seq % NR_BUFS] = true;   /* oznacz bufor jako pełny */
               in_buf[r.seq % NR_BUFS] = r.info;   /* wprowadź dane do bufora */
               while (arrived[frame_expected % NR_BUFS]) {
                  /* przekaż ramki i przesuń okno. */
                  to_network_layer(&in_buf[frame_expected % NR_BUFS]);
                  no_nak = true;
                  arrived[frame_expected % NR_BUFS] = false;
                  inc(frame_expected); /* podnieś dolną krawędź okna odbiornika */
                  inc(too_far);        /* podnieś górną krawędź okna odbiornika */
                  start_ack_timer();   /* czy potrzebne osobne potwierdzenie */
               }
            }
         }
```

```
            if((r.kind==nak) &&
            between(ack_expected,(r.ack+1)%(MAX_SEQ+1),next_frame_to_send))
            send_frame(data, (r.ack+1) % (MAX_SEQ + 1), frame_expected, out_buf);

            while (between(ack_expected, r.ack, next_frame_to_send)) {
               nbuffered = nbuffered - 1;   /* obsłuż doczepione potwierdzenie */
               stop_timer(ack_expected % NR_BUFS);   /* ramka dotarła nienaruszona */
               inc(ack_expected);           /* podnieś dolną krawędź okna nadajnika */
            }
            break;

         case cksum_err:
            if (no_nak) send_frame(nak, 0, frame_expected, out_buf); /* ramka uszkodzona
            */
            break;

         case timeout:
            send_frame(data, oldest_frame, frame_expected, out_buf);  /* upłynął nasz
            czas oczekiwania */
            break;

         case ack_timeout:
            send_frame(ack,0,frame_expected, out_buf);   /* wygasł czasomierz ack;
            wyślij potwierdzenie */
      }
      if (nbuffered < NR_BUFS) enable_network_layer(); else disable_network_layer();
   }
}
```

Odbieranie ramek poza kolejnością wprowadza pewne problemy nieobecne w protokołach, w których ramki są przyjmowane tylko we właściwej kolejności. Najłatwiej będzie zilustrować ten problem na przykładzie. Załóżmy, że mamy 3-bitowy numer sekwencyjny, więc nadajnik ma prawo wysłać do siedmiu ramek, zanim zostanie zmuszony do zatrzymania się i czekania na potwierdzenie. Początkowe okna nadawcze i odbiorcze wyglądają jak na rysunku 3.13 (a). Nadajnik wysyła teraz ramki o numerach od 0 do 6. Okno odbiornika pozwala mu przyjąć każdą ramkę o numerze sekwencyjnym od 0 do 6 włącznie. Wszystkie siedem ramek dociera poprawnie, więc odbiornik potwierdza je i przesuwa okno, aby pozwolić na przyjęcie ramki o numerze 7, 0, 1, 2, 3, 4 lub 5, jak na rysunku 3.13 (b). Wszystkie siedem buforów jest oznaczonych jako puste.



RYSUNEK 3.13. (a) Początkowa sytuacja z oknem o rozmiarze 7. (b) Sytuacja po wysłaniu i odebraniu siedmiu ramek, lecz przed ich potwierdzeniem. (c) Początkowa sytuacja z oknem o rozmiarze 4. (d) Sytuacja po wysłaniu i odebraniu czterech ramek, lecz przed ich potwierdzeniem

W tym momencie następuje katastrofa w postaci pioruna trafiającego w słup telefoniczny i niszczącego wszystkie potwierdzenia. W końcu upływa czas oczekiwania i nadajnik ponownie wysyła ramkę 0. Gdy ta dociera do odbiornika, przeprowadzana jest kontrola, czy mieści się w oknie odbiornika. Niestety, na rysunku 3.13 (b) ramka 0 mieści się w nowym oknie, więc zostanie przyjęta. Odbiornik odsyła „na barana" potwierdzenie ramki 6, ponieważ ramki od 0 do 6 zostały odebrane.

Nadajnik dowiaduje się z przyjemnością, że wszystkie wysłane ramki dotarły poprawnie, więc przesuwa swoje okno i natychmiast wysyła ramki nr 7, 0, 1, 2, 3, 4 i 5. Ramka 7 zostanie przyjęta przez odbiornik i jej pakiet zostanie przekazany bezpośrednio do warstwy sieciowej. Bezpośrednio po tym warstwa łącza danych odbiornika sprawdzi, czy ma już poprawną ramkę 0, odkryje, że tak, i przekaże zawarty w niej pakiet do warstwy sieciowej. Wobec tego warstwa sieciowa otrzyma niepoprawny pakiet i protokół zawiedzie.

Sedno problemu tkwi w tym, że po przesunięciu w górę okna przez odbiornik nowy zakres poprawnych numerów sekwencyjnych nałożył się na stary. Wobec tego następna porcja ramek może zawierać albo duplikaty (jeśli wszystkie potwierdzenia zostaną utracone), albo nowe ramki (jeśli potwierdzenia zostały odebrane). Biedny odbiornik nie ma jak rozróżnić tych dwóch przypadków.

Rozwiązanie polega na zapewnieniu, że po przesunięciu okna w górę przez odbiornik nie będzie pokrycia z poprzednim oknem. Aby to zagwarantować, maksymalny rozmiar okna powinien obejmować najwyżej połowę numerów sekwencyjnych, jak na rysunku 3.13 (c) i (d). Na przykład, jeśli na numery sekwencyjne używane są 4 bity, to numery będą mieścić się w zakresie od 0 do 15. W każdej chwili powinno zalegać najwyżej osiem niepotwierdzonych ramek. W ten sposób, jeśli odbiornik właśnie przyjął ramki od 0 do 7 i przesunął okno tak, by zezwolić na przyjęcie ramek od 8 do 15, to będzie mógł jednoznacznie odróżnić przychodzące retransmisje (0 do 7) od nowych ramek (8 do 15). Ogólnie mówiąc, rozmiar okna dla protokołu 6. powinien wynosić (`MAX_SEQ` + 1)/2. Wobec tego dla 3-bitowych numerów sekwencyjnych rozmiar okna wynosi 4.

Ciekawe może być pytanie, ile buforów musi mieć odbiornik? Pod żadnym warunkiem nie będzie akceptował ramek o numerach sekwencyjnych poniżej dolnej lub powyżej górnej krawędzi okna. Wobec tego liczba potrzebnych buforów jest równa rozmiarowi okna, a nie zakresowi numerów sekwencyjnych. W powyższym przykładzie z 4-bitowym numerem sekwencyjnym potrzebnych jest osiem buforów o numerach od 0 do 7. Gdy przychodzi ramka o numerze *i*, zostaje umieszczona w buforze (*i* mod 8). Proszę zwrócić uwagę, że chociaż *i* oraz [(*i* + 8) mod 8] „konkurują" o ten sam bufor, to nigdy jednocześnie nie znajdują się w oknie, ponieważ to wymagałoby okna o rozmiarze przynajmniej 9.

Z tego samego powodu liczba potrzebnych czasomierzy jest równa liczbie buforów, a nie liczbie numerów sekwencyjnych. W rzeczywistości z każdym buforem jest skojarzony czasomierz. Gdy upływa czas oczekiwania, zawartość bufora zostaje wysłana ponownie.

W protokole 5. zostało poczynione niejawne założenie, że kanał jest mocno obciążony. Gdy przybywa ramka, potwierdzenie nie jest wysyłane natychmiast, lecz dołączane „na barana" do następnej wychodzącej ramki danych. W razie niewielkiego ruchu w drugą stronę potwierdzenia będą wstrzymywane na długi czas. Jeśli w jednym kierunku odbywa się intensywny ruch, a w drugim nie ma żadnego, zostanie wysłanych tylko `MAX_SEQ` pakietów, a następnie protokół zablokuje się, dlatego też musieliśmy założyć, że w drugą stronę odbywa się jakiś ruch.

Ten problem został rozwiązany w protokole 6. Po przybyciu ramki danych zgodnie z sekwencją zostaje uruchomiony pomocniczy czasomierz przez `start_ack_timer`. Jeśli nie pojawi się żadna transmisja w drugą stronę przed upłynięciem jego czasu, zostanie wysłana odrębna ramka potwierdzająca. Przerwanie spowodowane przez czasomierz pomocniczy nosi nazwę zdarzenia `ack_timeout`. Przy takiej konfiguracji możliwy jest ruch danych w jednym kierunku, ponieważ brak ramek zwrotnych, na których mogą się „podwieźć" potwierdzenia, nie stanowi już problemu. Istnieje tylko jeden pomocniczy czasomierz, a jeśli procedura `start_ack_timer` zostanie wywołana w trakcie działania czasomierza, to zostanie on zresetowany do pełnego interwału potwierdzenia.

Czas oczekiwania skojarzony z pomocniczym czasomierzem powinien być wyraźnie krótszy od czasu używanego przy oczekiwaniu na potwierdzenie ramki danych. Ten warunek jest niezbędny, aby poprawnie odebrana ramka została potwierdzona na tyle szybko, by nie upłynął czas oczekiwania na potwierdzenie, powodując retransmisję ramki.

Protokół 6. do obsługi błędów używa bardziej efektywnej strategii niż protokół 5. Zawsze gdy odbiornik ma podstawy przypuszczać, że wystąpił błąd, wysyła ramkę z negatywnym potwierdzeniem (NAK) z powrotem do nadajnika. Taka ramka jest żądaniem retransmisji ramki wskazanej w NAK. Są dwie sytuacje, w których odbiornik może mieć podejrzenia: gdy dotarła ramka uszkodzona lub inna niż oczekiwana (ramka mogła zostać zgubiona). Aby uniknąć wielokrotnych żądań retransmisji tej samej utraconej ramki, odbiornik powinien pamiętać, czy wysłał już NAK dla danej ramki. Zmienna `no_nak` w protokole 6. ma wartość `true`, jeśli negatywne potwierdzenie nie zostało jeszcze wysłane dla `frame_expected`. Jeśli ramka NAK zostanie uszkodzona lub utracona, to nie stanie się nic poważnego, ponieważ w nadajniku w końcu upłynie czas oczekiwania i brakująca ramka zostanie i tak ponownie wysłana. Jeśli po wysłaniu i zgubieniu NAK przyjdzie niewłaściwa ramka, `no_nak` będzie miała wartość `true` i zostanie uruchomiony pomocniczy czasomierz. Gdy upłynie jego czas oczekiwania, zostanie wysłana ramka ACK do ponownej synchronizacji nadajnika z bieżącym stanem odbiornika.

W niektórych sytuacjach czas wymagany na propagację ramki do celu, przetworzenie jej tutaj i powrót potwierdzenia jest (niemal) stały. W takich warunkach nadajnik może ustawić swój czasomierz tak, że czas oczekiwania będzie tylko trochę dłuższy od standardowego okresu spodziewanego pomiędzy wysłaniem ramki i otrzymaniem jej potwierdzenia. Jednakże jeśli ten czas jest mocno zmienny, nadajnik ma możliwość albo ustawić niską wartość czasu oczekiwania (i ryzykować niepotrzebne retransmisje), albo dużą (i po wystąpieniu błędu na dłuższy czas wchodzić w stan bezczynności).

Obie opcje powodują marnowanie pasma. Jeśli ruch w drugą stronę jest sporadyczny, czas pomiędzy potwierdzeniami będzie nieregularny: krótszy w obecności ruchu w drugą stronę i dłuższy przy jego braku. Zmienny czas przetwarzania w odbiorniku również może stanowić tu problem. Ogólnie mówiąc, gdy odchylenie standardowe interwału potwierdzenia jest małe w porównaniu z samym interwałem, czasomierz może być ustawiony z małym zapasem i ramki NAK nie są przydatne. W przeciwnym razie czasomierz musi zostać ustawiony z dużym zapasem aby uniknąć niepotrzebnych retransmisji, lecz NAK mogą znacząco przyspieszyć retransmisję utraconych i uszkodzonych ramek.

Ściśle powiązana z okresami oczekiwania i potwierdzeniami negatywnymi jest kwestia ustalenia, która ramka spowodowała upłynięcie czasu oczekiwania. W protokole 5. jest to zawsze `ack_expected`, ponieważ jest to zawsze najstarsza ramka. W protokole 6. nie istnieje żaden prosty sposób ustalenia sprawcy. Załóżmy, że zostały wysłane ramki od 0 do 4, co oznacza, że lista zaległych ramek zawiera 01234, od najstarszej do najnowszej. Wyobraźmy sobie teraz, że upłynie czas oczekiwania dla 0, zostanie wysłana nowa ramka (5), dla ramek 1 i 2 upłyną czasy oczekiwania i zostanie wysłana jeszcze jedna nowa ramka (6). W tym momencie lista zaległych ramek wygląda tak: 3405126, od najstarszej do najnowszej. Jeśli przez jakiś czas będzie utracony wszelki ruch przychodzący (czyli ramki niosące potwierdzenia), w tej kolejności upłyną czasy oczekiwania dla siedmiu zaległych ramek.

Aby nie komplikować przykładu jeszcze bardziej, nie pokażemy zarządzania czasomierzami. Zamiast tego założymy, że po upłynięciu czasu oczekiwania ustawiana jest zmienna `oldest_frame` w celu wskazania, dla której ramki upłynął czas oczekiwania.

3.5. Weryfikacja protokołów

Rzeczywiste protokoły i implementujące je programy są często dość skomplikowane. Wobec tego sporo badań poświęcono próbom znalezienia formalnych, matematycznych technik specyfikowania i weryfikacji protokołów. W poniższych punktach przyjrzymy się kilku modelom i technikom. Wprawdzie spojrzymy na nie w kontekście warstwy łącza danych, lecz stosują się one również do innych warstw.

3.5.1. Modele oparte na automatach skończonych

Kluczowym konceptem stosowanym w wielu modelach protokołów jest **automat skończony** (inaczej **automat o skończonej liczbie stanów**). W tej technice każdy automat protokołu (np. nadajnik lub odbiornik) w każdej chwili znajduje się w określonym stanie. Jego stan składa się z wszystkich wartości zmiennych, łącznie z licznikiem rozkazów.

W większości przypadków dużą liczbę stanów można na potrzeby analizy zgrupować razem. Na przykład dla odbiornika z protokołu 3. możemy wyabstrahować z wszystkich możliwych stanów dwa istotne: czekanie na ramkę 0 i czekanie na ramkę 1. Wszystkie pozostałe stany mogą być uważane za przejściowe, jako jedynie kroki po drodze do jednego z głównych stanów. Zwykle stany wybierane są w tych momentach, gdy automat protokołu czeka na wystąpienie kolejnego zdarzenia, np. w naszych przykładach wykonuje wywołanie procedury `wait(event)`. W tym miejscu stan automatu protokołu w pełni określają stany jego zmiennych. Wobec tego liczba stanów wynosi 2^n, gdzie n jest liczbą bitów potrzebnych do reprezentowania wszystkich zmiennych razem.

Stan całego systemu jest kombinacją wszystkich stanów obu automatów protokołu i kanału. Stan kanału jest określany przez jego zawartość. Jeśli ponownie jako przykład weźmiemy protokół 3., to kanał ma cztery możliwe stany: ramka 0 lub 1 przechodząca z nadajnika do odbiornika, ramka potwierdzająca przechodząca w drugą stronę oraz kanał pusty. Jeśli zamodelujemy nadajnik i odbiornik jako mające po dwa stany, kompletny system będzie miał 16 różnych stanów.

Warto powiedzieć tu kilka słów o stanie kanału. Idea ramki znajdującej się „w kanale" jest oczywiście abstrakcją. Używając tego pojęcia, mamy na myśli, że ramka mogła zostać odebrana, lecz nie została przetworzona u celu. Ramka pozostaje „w kanale", dopóki automat protokołu nie wykona `from_physical_layer` i nie przetworzy jej.

Z każdego stanu istnieje zero lub więcej możliwych **przejść** (inaczej **zmian stanów**) do innych stanów. Przejścia odbywają się, gdy występuje jakieś zdarzenie. Dla automatu protokołu zmiana stanu może się odbyć w przypadku wysłania ramki, odebrania ramki, upłynięcia czasu oczekiwania, wystąpienia przerwania itp. Typowymi zdarzeniami dla kanału są: wprowadzenie nowej ramki do kanału przez automat protokołu, doręczenie ramki do automatu protokołu lub utrata ramki z powodu zakłóceń. Mając pełny opis automatów protokołu i właściwości kanału, możemy narysować graf skierowany przedstawiający wszystkie stany jako węzły, a zmiany stanów w postaci krawędzi skierowanych.

Jeden specjalny stan zostaje wyznaczony jako **stan początkowy**. Odpowiada on opisowi systemu zaraz po rozpoczęciu pracy lub w jakimś wygodnym miejscu wkrótce potem. Ze stanu początkowego część innych stanów, a być może wszystkie, można osiągnąć przez sekwencję zmian stanów. Stosując dobrze znane techniki z teorii grafów (np. obliczanie przechodniego domknięcia grafu), możemy ustalić, które stany są dostępne, a które nie. Ta technika nosi nazwę **analizy osiągalności** (Lin i in., 1987). Ta analiza może pomóc w ustaleniu, czy protokół jest poprawny.

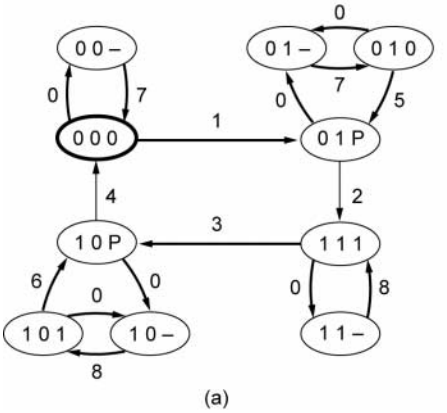
Formalnie model protokołu w automatach skończonych możemy uznać za czwórkę (S, M, I, T), gdzie:

S jest zbiorem stanów, w których mogą być procesy i kanał.
M jest zbiorem ramek, które mogą być przesyłane kanałem.
I jest zbiorem stanów początkowych procesów.
T jest zbiorem przejść pomiędzy stanami.

Na początku wszystkie procesy są w swoich stanach początkowych. Następnie zaczynają zachodzić zdarzenia, na przykład pojawiają się ramki dostępne do przesłania lub uruchamiają się czasomierze. Każde zdarzenie może spowodować podjęcie akcji przez jeden z procesów lub kanał i przejście do nowego stanu. Dokładne wypisanie wszystkich możliwych następców każdego stanu pozwoli zbudować graf osiągalności i przeanalizować protokół.

Analiza osiągalności umożliwia wykrycie różnorodnych błędów w specyfikacji protokołu. Na przykład, jeśli pojawi się określona ramka w określonym stanie i automat skończony nie powie, jaka akcja ma zostać podjęta, to specyfikacja jest błędna (niekompletna). Jeśli istnieje zbiór stanów, z których nie ma wyjścia i nie może wystąpić postęp (tzn. nie będzie można odbierać dalszych poprawnych ramek), to mamy inny typ błędu — zakleszczenie. Mniej poważnym błędem jest specyfikacja protokołu mówiąca, jak obsłużyć zdarzenie w stanie, w którym to zdarzenie nie może wystąpić (obce przejście). Inne błędy również mogą zostać wykryte.

Jako przykład modelu automatów skończonych weźmy rysunek 3.14 (a). Graf ten odpowiada protokołowi 3. zgodnie z wcześniejszym opisem: każdy automat protokołu ma dwa stany, a kanał cztery. Istnieje łącznie 16 stanów, z których nie wszystkie są osiągalne z początkowego. Stany nieosiągalne nie zostały przedstawione na rysunku. Dla uproszczenia pominięte są też błędy sumy kontrolnej.



(a)

Przejście	Nadajnik czy odbiornik?	Ramka przyjęta	Ramka wysłana	Do warstwy sieciowej?
0	–	(ramka utracona)		–
1	O	0	P	Tak
2	N	P	1	–
3	O	1	P	Tak
4	N	P	0	–
5	O	0	P	Nie
6	O	1	P	Nie
7	N	(timeout)	0	–
8	N	(timeout)	1	–

(b)

RYSUNEK 3.14. (a) Diagram stanów dla protokołu 3. (b) Przejścia

Każdy stan jest oznaczany trzema literami, *SRC*, gdzie *S* jest równe 0 lub 1 zależnie od ramki, którą nadajnik próbuje wysłać; *R* również jest równe 0 lub 1 i odpowiada ramce oczekiwanej przez odbiornik, a *C* ma wartość 0, 1, A lub pustą (–) odpowiadającą stanowi kanału. W powyższym przykładzie wybraliśmy stan początkowy (000) — inaczej mówiąc, nadajnik wysłał właśnie ramkę 0, odbiornik oczekuje ramki 0 i w kanale znajduje się ramka 0.

Na rysunku 3.14 widać dziewięć typów przejść. Przejście 0 zawiera utratę zawartości przez kanał. Przejście 1 zawiera poprawne doręczenie pakietu 0 do odbiornika, po którym odbiornik zmienia swój stan na oczekiwanie ramki 1 i wysyła potwierdzenie. Przejście 1 odpowiada też doręczeniu przez odbiornik pakietu 0 do warstwy sieciowej. Pozostałe przejścia przedstawia rysunek 3.14 (b). Przyjście ramki z błędem sumy kontrolnej nie zostało pokazane, ponieważ nie zmienia stanu (w protokole 3.).

Podczas normalnego działania przejścia 1, 2, 3 i 4 są kolejno powtarzane raz za razem. W każdym cyklu zostają doręczone dwa pakiety, przywracając nadajnik do pierwotnego stanu próby wysłania nowej ramki z numerem sekwencyjnym 0. Jeśli kanał utraci ramkę 0, przechodzi ze stanu (000) do (00–). W końcu upływa czas oczekiwania nadajnika (przejście 7), i system wraca do (000). Utrata potwierdzenia jest bardziej skomplikowana i do naprawy szkody wymaga dwóch przejść, 7 i 5 lub 8 i 6.

Jedną z właściwości, jaką musi mieć protokół z 1-bitowym numerem sekwencyjnym, jest ta, że niezależnie od sekwencji zdarzeń, jaka może wystąpić, nadajnik nigdy nie może doręczyć dwóch parzystych pakietów pod rząd bez rozdzielenia ich nieparzystym i vice versa. Z grafu na rysunku 3.14 widzimy, że to stwierdzenie można bardziej formalnie sformułować tak: „nie może

istnieć żadna ścieżka ze stanu początkowego, na której występują dwa wystąpienia przejścia 1 bez wystąpienia pomiędzy nimi przejścia 3 lub vice versa". Z rysunku widać, że pod tym względem protokół jest poprawny.

Jest jeszcze inny podobny wymóg, że nie może istnieć żadna ścieżka, na której stan nadajnika zmienia się dwukrotnie (np. z 0 na 1 i z powrotem na 0), podczas gdy stan odbiornika pozostaje stały. Gdyby taka ścieżka istniała, to w odpowiadającej jej sekwencji ramek dwie ramki zostałyby nieodwracalnie utracone bez zauważenia tego przez odbiornik. Doręczona sekwencja pakietów zawierałaby niewykrytą lukę o długości dwóch pakietów.

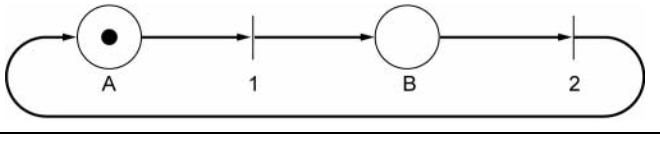
Jeszcze jedną ważną właściwością protokołu jest brak zakleszczeń. **Zakleszczenie** (ang. *deadlock*) oznacza sytuację, w której protokół nie może pójść dalej (np. doręczać pakietów do warstwy sieciowej), niezależnie od tego, jaka sekwencja zdarzeń wystąpi. W modelu grafu zakleszczenie charakteryzuje się obecnością podzbioru stanów osiągalnego ze stanu początkowego i mającego dwie właściwości:

(1) Nie istnieje przejście wychodzące z podzbioru.
(2) Nie istnieją przejścia w podzbiorze powodujące postęp w przód.

Po wejściu do stanu zakleszczenia protokół pozostaje w nim na zawsze. Ponownie łatwo zobaczyć z naszego grafu, że w protokole 3. nie występują zakleszczenia.

3.5.2. Modele sieci Petriego

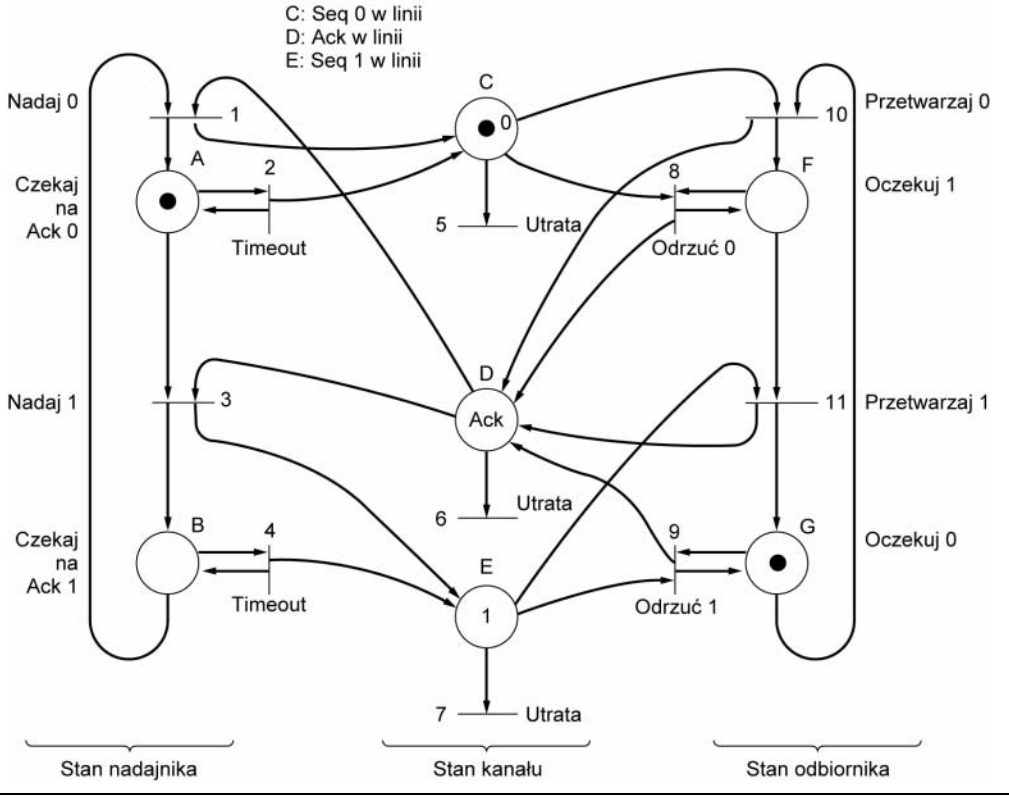
Automaty skończone nie są jedyną techniką pozwalającą formalnie specyfikować protokoły. W tym punkcie opiszemy zupełnie odmienną technikę zwaną siecią Petriego (Danthine, 1980). Sieć Petriego ma cztery podstawowe składniki: miejsca, przejścia, łuki i żetony. **Miejsce** reprezentuje stan, w którym może znajdować się system lub jego część. Rysunek 3.15 przedstawia sieć Petriego z dwoma miejscami, *A* i *B*, reprezentowanymi przez okręgi. System znajduje się obecnie w stanie A, na co wskazuje **żeton** (tłusta kropka) w miejscu *A*. Przejście jest oznaczane pionową lub poziomą poprzeczką. Każde przejście ma zero lub więcej **łuków wejściowych** przychodzących ze swoich miejsc wejściowych, oraz zero lub więcej **łuków wyjściowych**, prowadzących do miejsc wyjściowych.



RYSUNEK 3.15. Sieć Petriego z dwoma miejscami i dwoma przejściami

Przejście jest **aktywne**, jeśli w każdym z jego miejsc wejściowych znajduje się przynajmniej jeden żeton wejściowy. Aktywne przejście może odpalić w dowolnej chwili, usuwając jeden żeton z każdego miejsca wejściowego i umieszczając żeton w każdym miejscu wyjściowym. Jeśli liczba łuków wyjściowych różni się od liczby łuków wejściowych, żetony nie są zachowywane. Jeśli aktywne są dwa przejścia lub więcej, odpalić może dowolne z nich. Wybór przejścia do odpalenia jest nieokreślony, dlatego też sieci Petriego są przydatne do modelowania protokołów. Sieć Petriego z rysunku 3.15 jest deterministyczna i może służyć do modelowania dowolnego dwufazowego procesu (np. zachowania noworodka: jedz, śpij, jedz, śpij, jedz, śpij i tak dalej). Podobnie jak w każdym narzędziu modelującym niepotrzebne szczegóły zostały tu pominięte.

Rysunek 3.16 przedstawia model sieci Petriego dla listingu 3.4. W przeciwieństwie do modelu z automatami skończonymi nie ma w nim stanów złożonych: stany nadajnika, kanału i odbiornika są



RYSUNEK 3.16. Model sieci Petriego dla protokołu 3.

reprezentowane oddzielnie. Przejścia 1 i 2 odpowiadają transmisji ramki 0 przez nadajnik i upłynięciu czasu oczekiwania. Przejścia 3 i 4 oznaczają to samo dla ramki 1. Przejścia 5, 6 i 7 odpowiadają kolejno utracie ramki 0, potwierdzenia i ramki 1. Przejścia 8 i 9 występują, gdy do odbiornika dotrze ramka danych z niewłaściwym numerem sekwencyjnym. Przejścia 10 i 11 reprezentują dotarcie do odbiornika następnej ramki w sekwencji i jej doręczenie do warstwy sieciowej.

Sieci Petriego mogą służyć do wykrywania błędów w protokole podobnie jak automaty skończone. Na przykład gdyby jakaś sekwencja odpaleń zawierała przejście 10 dwukrotnie bez przejścia 11 pomiędzy nimi, to protokół byłby niepoprawny. Pojęcie zakleszczenia w sieci Petriego jest podobne do swojego odpowiednika w automacie skończonym.

Sieci Petriego mogą być reprezentowane w wygodniej formie algebraicznej przypominającej gramatykę. Każde przejście dodaje jedną regułę do gramatyki. Każda reguła określa miejsca wejściowe i wyjściowe dla przejścia. Ponieważ rysunek 3.16 zawiera 11 przejść, to jego gramatyka zawiera 11 reguł ponumerowanych od 1 do 11, z których każda odpowiada przejściu o tym samym numerze. Gramatyka dla sieci Petriego z rysunku 3.16 jest następująca:

1: $BD \rightarrow AC$
2: $A \rightarrow A$
3: $AD \rightarrow BE$
4: $B \rightarrow B$

5: C →
6: D →
7: E →
8: CF → DF
9: EG → DG
10: CF → DF
11: EF → DG

To ciekawe, jak udało nam się zredukować złożony protokół do 11 reguł gramatycznych łatwych do manipulowania przez programy komputerowe.

Bieżący stan sieci Petriego jest reprezentowany jako nieuporządkowany zbiór miejsc, z których każde miejsce jest tyle razy reprezentowane w zbiorze, ile ma żetonów. Każda reguła, dla której obecne są wszystkie miejsca po lewej, może zostać odpalona, co powoduje usunięcie tych miejsc ze stanu bieżącego i dodanie ich miejsc wyjściowych do stanu bieżącego. Oznaczone dla rysunku 3.16 są *ACG* (tzn. *A*, *C* i *G* mają po jednym żetonie). Wobec tego reguły 2., 5. i 10. są aktywne i każda z nich może zostać zastosowana, prowadząc do nowego stanu (być może z takim samym cechowaniem jak poprzedni). Z drugiej strony reguła 3. ($AD \rightarrow BE$) nie może zostać zastosowana, ponieważ *D* nie jest oznaczony.

3.6. Przykładowe protokoły łącza danych

W poniższych punktach opiszemy kilka powszechnie stosowanych protokołów łącza danych. Pierwszy z nich, HDLC, jest klasycznym protokołem o organizacji bitowej, którego odmiany były przez dziesięciolecia używane w różnych zastosowaniach. Drugi protokół łącza danych, PPP, służy do łączenia komputerów domowych z Internetem.

3.6.1. Protokół HDLC

W tym punkcie przyjrzymy się grupie blisko spokrewnionych protokołów, które są dość stare, lecz nadal powszechnie stosowane. Wszystkie wywodzą się z protokołu łącza danych użytego po raz pierwszy w świecie komputerów mainframe firmy IBM — **SDLC** (**Synchronous Data Link Control**). Po opracowaniu SDLC IBM zgłosił ten protokół do ANSI i ISO w celu zaakceptowania jako standardu, odpowiednio w USA i międzynarodowego. ANSI zmodyfikował go do wersji **ADCCP** (**Advanced Data Communication Control Procedure**), a ISO do wersji **HDLC** (**High-level Data Link Control**). Następnie CCITT przyjął i zmodyfikował HDLC na swój **LAP** (**Link Access Procedure**) jako element standardu interfejsu sieciowego X.25, lecz później zmodyfikował jeszcze raz do **LAPB**, aby zapewnić większą zgodność z późniejszymi wersjami HDLC. Miło mieć tak dużo standardów do wyboru, nieprawdaż? Co więcej, jeśli żaden z nich nam się nie spodoba, możemy po prostu zaczekać rok na nowszy model.

Te protokoły opierają się na tych samych zasadach. Wszystkie mają organizację bitową i wszystkie stosują wypełnianie bitami dla przezroczystości danych. Różnią się tylko szczegółami, aczkolwiek w irytujący sposób. Przedstawiony poniżej opis protokołów zorganizowanych bitowo jest pomyślany jako ogólne wprowadzenie. Konkretne szczegóły poszczególnych protokołów dostępne są w ich definicjach.

Wszystkie protokoły zorganizowane bitowo używają struktury ramki jak na rysunku 3.17. Pole *Adres* jest ważne przede wszystkim na liniach z wieloma terminalami, gdzie służy do zidentyfikowania jednego z terminali. W łączach dwupunktowych czasem jest używane do odróżniania poleceń od odpowiedzi.

RYSUNEK 3.17. Format ramki w protokołach zorganizowanych bitowo

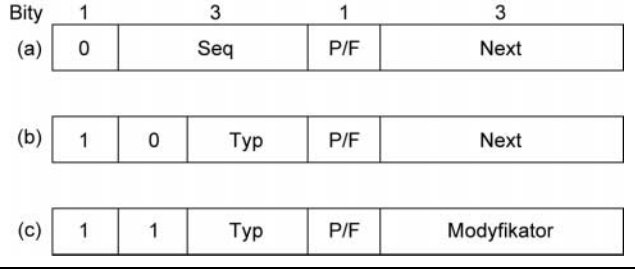
Pole *Sterowanie* jest przeznaczone na numery sekwencyjne, potwierdzenia i inne zastosowania omówione poniżej.

Pole *Dane* może zawierać dowolne informacje. Jego długość może być dowolna, aczkolwiek skuteczność sumy kontrolnej maleje ze wzrostem długości ramki z uwagi na prawdopodobieństwo wystąpienia więcej niż jednej paczki błędów.

Pole *Suma kontrolna* zawiera kod CRC stosujący technikę, którą omówiliśmy w punkcie 3.2.2.

Ramka jest ograniczona z obu stron sekwencją znacznikową 0111110. W bezczynnych liniach dwupunktowych sekwencje znacznikowe są nadawane w sposób ciągły. Minimalna ramka zawiera trzy pola o łącznej długości 32 bitów, nie licząc znaczników na końcach.

Istnieją trzy rodzaje ramek: **Information** (ramka danych), **Supervisory** (ramka nadzorcza) i **Unnumbered** (ramka nienumerowana). Zawartości pola *Sterowanie* dla tych trzech typów przedstawia rysunek 3.18. Protokół używa okna przesuwnego z 3-bitowym numerem sekwencyjnym. W każdej chwili może być zaległych najwyżej siedem niepotwierdzonych ramek. Pole *Seq* z rysunku 3.18 (a) zawiera numer sekwencyjny ramki. Pole *Next* jest potwierdzeniem transportowanym „na barana". Jednakże wszystkie protokoły stosują się do konwencji, w której zamiast numeru ostatniej poprawnie otrzymanej ramki w tym polu przesyłany jest numer pierwszej ramki jeszcze nieodebranej (tzn. następnej oczekiwanej). Wybór pomiędzy ostatnią ramką odebraną i następną oczekiwaną jest arbitralny; nie jest ważne, która konwencja jest używana, pod warunkiem, że jest używana konsekwentnie.



RYSUNEK 3.18. Pola sterujące: (a) ramki danych, (b) ramki nadzorczej, (c) ramki nienumerowanej

Bit P/F oznacza *Poll/Final*. Jest on używany, gdy komputer (lub koncentrator) odpytuje grupę terminali. Gdy jest używany jako P, komputer zaprasza jeden terminal do wysłania danych. Wszystkie ramki nadawane przez terminal, z wyjątkiem ostatniej, mają bit P/F ustawiony na P, a ostatnia na F.

W niektórych protokołach bit P/F służy do zmuszania drugiego komputera do natychmiastowego wysłania ramki nadzorczej, zamiast czekać na transmisję w drugą stronę, do której mógłby doczepić informacje okna. Ten bit ma też pewne pomniejsze zastosowania w związku z ramkami nienumerowanymi.

Różne typy ramek nadzorczych są rozróżniane przez pole *Typ*. Typ 0 oznacza ramkę potwierdzającą (oficjalnie nazywaną RECEIVE READY) służącą do wskazania następnej oczekiwanej ramki. Ta ramka jest używana wtedy, gdy nie ma transmisji w drugą stronę, do której można byłoby doczepić potwierdzenie.

Typ 1 jest ramką potwierdzenia negatywnego (oficjalnie nazywaną REJECT). Służy do wskazania, że został wykryty błąd transmisji. Pole *Next* wskazuje pierwszą ramkę w sekwencji nieodebraną poprawnie (tzn. ramkę wymagającą retransmisji). Od nadajnika wymaga się retransmisji wszystkich zaległych ramek, poczynając od *Next*. Ta strategia jest bardziej podobna do naszego protokołu nr 5 niż do protokołu nr 6.

Typ 2 to RECEIVE NOT READY. Potwierdza wszystkie ramki aż do *Next* za jej wyjątkiem, podobnie jak RECEIVE NOT READY, lecz każe nadajnikowi wstrzymać nadawanie. RECEIVE NOT READY służy do sygnalizacji chwilowych problemów z odbiornikiem, takich jak brak wolnych buforów, lecz jest alternatywą dla sterowania oknem przesuwnym. Po naprawie sytuacji odbiornik wysyła RECEIVE READY, REJECT lub określone ramki sterujące.

Typ 3 to SELECTIVE REJECT, żądający retransmisji tylko podanej ramki. W tym sensie przypomina bardziej nasz protokół nr 6 niż nr 5, więc jest najbardziej przydatny wtedy, gdy rozmiar okna nadajnika jest równy połowie przestrzeni numerów sekwencyjnych lub mniejszy. Jeśli więc odbiornik chce buforować ramki spoza kolejności, aby móc ich ewentualnie użyć później, to może wymusić retransmisję dowolnej konkretnej ramki, używając SELECTIVE REJECT. HDLC i ADCCP pozwalają na ten typ ramki, lecz SDLC i LAPB nie dopuszczają go, i w tych protokołach typ 3 jest niezdefiniowany.

Trzecią klasą ramek są ramki nienumerowane (*Unnumbered*). Czasem są używane do celów sterowania, lecz mogą również przenosić dane, gdy potrzebna jest zawodna usługa bezpołączeniowa. Różne protokoły o organizacji bitowej różnią się tu znacząco, w przeciwieństwie do dwóch pozostałych typów, które są w nich niemal identyczne. Do oznaczenia typu ramki dostępnych jest 5 bitów, lecz nie wszystkie 32 możliwości zostały wykorzystane.

Wszystkie protokoły udostępniają polecenie *DISC* (*DISConnect* — rozłącz) pozwalające komputerowi ogłosić, że będzie wyłączony (na przykład do konserwacji prewencyjnej). Mają też polecenie, które pozwala właśnie podłączonemu komputerowi ogłosić swoją obecność i wymusić wyzerowanie wszystkich numerów sekwencyjnych. To polecenie nosi nazwę *SNRM* (*Set Normal Response Mode* — ustaw normalny tryb odpowiedzi). Niestety, ten „normalny tryb odpowiedzi" jest daleki od normalności. Jest to tryb niezrównoważony (tzn. asymetryczny), w którym jeden koniec łącza jest nadrzędny, a drugi podrzędny. SNRM pochodzi z czasów, gdy komunikacja danych oznaczała komunikację niemego terminala z dużym komputerem-hostem, ewidentnie asymetryczną. Aby protokół lepiej nadawał się do komunikacji pomiędzy partnerami równorzędnymi, HDLC i LAPB zawierają dodatkowe polecenie *SABM* (*Set Asynchronous Balanced Mode* — ustaw asynchroniczny tryb zrównoważony), które zeruje linię i deklaruje obie strony jako równorzędne. Dostępne są też polecenia SABME i SNRME, takie same jak odpowiednio SABM i SNRM, z tą różnicą, że pozwalają na rozszerzony format ramki używający 7-bitowych numerów sekwencyjnych zamiast 3-bitowych.

Trzecim poleceniem udostępnianym przez wszystkie protokoły jest *FRMR* (*FRaMe Reject*) służące do wskazania, że została odebrana ramka z poprawną sumą kontrolną, lecz o niedopuszczalnej składni. Przykładami takiej składni może być typ 3 ramki nadzorczej w LAPB, ramka krótsza niż 32 bity, nielegalna ramka sterująca, potwierdzenie ramki spoza okna itp. Ramki FRMR zawierają 24-bitowe pole danych informujące, co jest nie tak z ramką. W danych mieści się pole sterujące błędnej ramki, parametry okna i zbiór bitów służących do sygnalizacji konkretnych błędów.

Ramki sterujące mogą zostać utracone lub uszkodzone, więc również muszą być potwierdzane. Do tego celu służy specjalna ramka sterująca, zwana *Unnumbered Acknowledgement* (potwierdzenie nienumerowane). Ponieważ tylko jedna ramka sterująca może być zaległa, nigdy nie ma niejasności, która ramka zostaje potwierdzona.

Pozostałe ramki sterujące są związane z inicjalizacją, odpytywaniem i raportowaniem stanu. Istnieje też ramka sterująca, która może zawierać dowolne informacje — *UI* (*Unnumbered Information*). Te dane nie są przekazywane do warstwy sieciowej, lecz są przeznaczone dla samej warstwy łącza danych w odbiorniku.

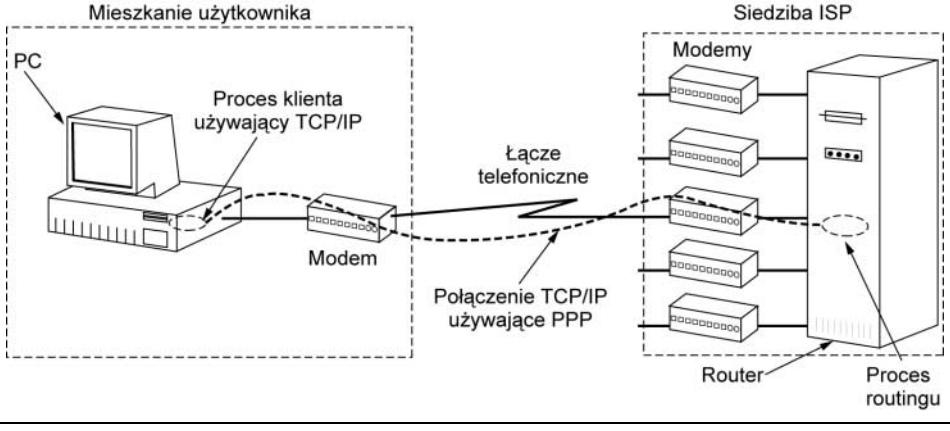
Mimo swojej popularności HDLC jest daleki od doskonałości. Omówienie szeregu związanych z nim problemów przedstawia Fiorini i inni (1994).

3.6.2. Warstwa łącza danych w Internecie

Internet składa się z indywidualnych urządzeń (hostów i routerów) oraz łączącej je infrastruktury komunikacyjnej. W obrębie pojedynczego budynku do łączenia urządzeń powszechnie stosuje się sieci LAN, lecz większość infrastruktury rozległej opiera się na dwupunktowych łączach dzierżawionych. W rozdziale 4. opiszemy sieci LAN; tutaj przyjrzymy się protokołom łącza danych używanych w Internecie w łączach dwupunktowych.

W praktyce komunikacja dwupunktowa jest stosowana głównie w dwóch sytuacjach. Po pierwsze, tysiące organizacji dysponują jedną lub wieloma sieciami LAN, z których każda łączy pewną liczbę hostów (komputery osobiste, stacje robocze, serwery itd.) z routerem (lub mostem o podobnej funkcjonalności). Routery często są ze sobą połączone szkieletową siecią LAN. Wszelkie połączenia ze światem zewnętrznym przechodzą zwykle przez dwa lub więcej routerów mających dwupunktowe połączenia liniami dzierżawionymi z odległymi routerami. Właśnie te routery i ich linie dzierżawione składają się na podsieci komunikacyjne, na których opiera się Internet.

Drugim miejscem, w którym linie dwupunktowe grają w Internecie ważną rolę, są miliony domowych połączeń z Internetem używających modemów i linii telefonicznych. Zwykle użytkownik domowego komputera PC łączy się modemem z routerem dostawcy usług internetowych, po czym zaczyna działać jak prawdziwy host internetowy. Ta metoda działania różni się od linii dzierżawionej pomiędzy PC i routerem tylko tym, że połączenie zostaje zerwane po zakończeniu sesji przez użytkownika. Rysunek 3.19 przedstawia domowy PC łączący się z dostawcą usług internetowych. Modem został przedstawiony na zewnątrz komputera, aby podkreślić jego rolę, lecz nowoczesne komputery zawierają modemy wewnętrzne.



RYSUNEK 3.19. Komputer domowy grający rolę hosta internetowego

Zarówno w łączach między routerami, używających linii dzierżawionych, jak i w połączeniach liniami telefonicznymi pomiędzy hostem i routerem, niezbędny jest jakiś protokół łącza danych do ramkowania, kontroli błędów i innych zadań warstwy łącza danych, które omówiliśmy w tym rozdziale. Protokół używany w Internecie nosi nazwę PPP. Przyjrzyjmy mu się.

Protokół PPP

Internet potrzebuje protokołu dwupunktowego do wielu celów, w tym do transmisji pomiędzy routerami oraz do transmisji pomiędzy użytkownikiem domowym a ISP. Jest nim protokół **PPP** (**Point-to-Point Protocol**) zdefiniowany w RFC 1661 i rozwinięty w kilku innych RFC (np. 1662 i 1663). PPP wykrywa błędy, obsługuje szereg różnych protokołów, pozwala na negocjację adresów IP w momencie podłączenia, umożliwia uwierzytelnianie i ma wiele innych możliwości.
PPP udostępnia trzy funkcje:

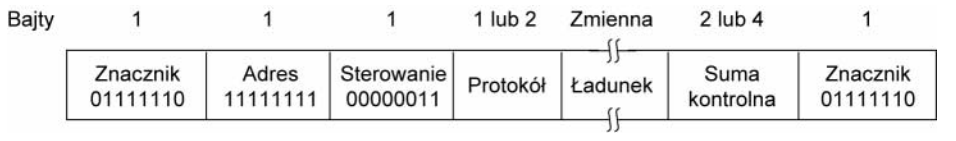
(1) Metodę ramkowania, która w sposób jednoznaczny wyznacza koniec jednej ramki i początek następnej. Format ramki obsługuje też wykrywanie błędów.
(2) Protokół sterowania łączem do uruchamiania i testowania linii, negocjowania opcji i wyłączania linii w sposób kontrolowany, gdy przestają być potrzebne. Ten protokół nosi nazwę **LCP** (**Link Control Protocol**) i obsługuje obwody synchroniczne i asynchroniczne oraz kodowanie bitowe i bajtowe.
(3) Sposób negocjacji opcji warstwy sieciowej niezależny od protokołu warstwy sieciowej, który będzie użyty. Wybrana metoda polega na użyciu innego **NCP** (**Network Control Protocol** — protokołu sterowania siecią) dla każdej obsługiwanej warstwy sieciowej.

Aby zobaczyć, jak to wszystko łączy się w całość, rozważmy typowy scenariusz, w którym użytkownik domowy łączy się z ISP w celu uczynienia ze swojego domowego PC tymczasowego hosta internetowego. Najpierw komputer PC łączy się za pomocą modemu z routerem. Po odebraniu połączenia przez router i nawiązaniu połączenia fizycznego PC wysyła do routera serię pakietów LCP w polu treści zasadniczej jednej lub więcej ramek PPP. Te pakiety i odpowiedzi na nie wybierają parametry PPP, które będą używane.

Po uzgodnieniu parametrów zostaje wysłana seria pakietów NCP do skonfigurowania warstwy sieciowej. Zwykle PC używa stosu protokołów TCP/IP, więc potrzebuje adresu IP. Dostępnych adresów IP jest za mało dla wszystkich, więc zwykle każdy dostawca usług internetowych otrzymuje pulę takich adresów i przydziela je dynamicznie do każdego świeżo podłączonego PC na czas sesji logowania. Jeśli dostawca posiada *n* adresów, to może obsłużyć najwyżej *n* równocześnie zalogowanych komputerów, lecz łączna baza klientów może być wielokrotnie większa. NCP dla IP przypisuje adres IP.

Teraz PC jest już hostem internetowym i może wysyłać i odbierać pakiety IP, podobnie jak hosty podłączone na sztywno. Gdy użytkownik skończy pracę, NCP zrywa połączenie w warstwie sieciowej i zwalnia adres IP. Następnie LCP zamyka połączenie w warstwie łącza danych. Na koniec komputer rozkazuje modemowi rozłączyć linię telefoniczną, zwalniając połączenie w warstwie fizycznej.

Format ramki PPP został tak dobrany, że w dużym stopniu przypomina format ramki HDLC, ponieważ nie ma sensu wyważać otwartych drzwi. Główna różnica pomiędzy PPP i HDLC polega na tym, że PPP jest zorganizowany znakowo, a nie bitowo. Przede wszystkim PPP w modemowych łączach telefonicznych używa napełniania bajtami, dzięki czemu każda ramka składa się z całkowitej liczby bajtów. Nie można wysłać ramki złożonej z 30,25 bajta jak w HDLC. Ramki PPP nie tylko mogą być wysyłane łączami telefonicznymi, lecz również poprzez SONET i przez prawdziwe zorganizowane bitowo linie HDLC (np. w połączeniach między routerami). Rysunek 3.20 przedstawia format ramki PPP.



RYSUNEK 3.20. Format pełnej ramki PPP dla pracy w trybie nienumerowanym

Wszystkie ramki PPP zaczynają się od standardowego bajta znacznikowego HDLC (01111110), który jest napełniany bajtowo, jeśli wystąpi w polu treści zasadniczej. Następne jest pole adresu (Address), zawsze zawierające wartość binarną 11111111 wskazującą, że wszystkie stacje powinny przyjąć ramkę. Użycie tej wartości pozwala uniknąć problemu z przydzielaniem adresów łącza danych.

Po polu adresu następuje pole sterujące (*Control*), którego domyślną wartością jest 00000011. Ta wartość oznacza ramkę nienumerowaną. Inaczej mówiąc, PPP domyślnie nie zapewnia niezawodnej transmisji z użyciem numerów sekwencyjnych i potwierdzeń. W środowiskach z dużym poziomem zakłóceń, na przykład w sieciach bezprzewodowych, można stosować niezawodną transmisję z użyciem trybu numerowanego. Szczegóły przedstawia RFC 1663, lecz w praktyce ten tryb jest rzadko używany.

Ponieważ pola adresu i sterujące w domyślnej konfiguracji są zawsze stałe, LCP udostępnia mechanizmy niezbędne do wynegocjowania przez obie strony opcji ich pominięcia i zaoszczędzenia 2 bajtów na ramkę.

Czwarte pole PPP jest polem protokołu. Jego zadaniem jest informowanie, jaki typ pakietu znajduje się w polu ładunku. Zostały zdefiniowane kody dla LCP, NCP, IP, IPX, AppleTalk i innych protokołów. Kody protokołów zaczynające się od bitu 0 to protokoły warstwy sieciowej, takie jak IP, IPX, OSI, CLNP i XNS. Zaczynające się od bitu 1 służą do negocjowania innych protokołów. Zaliczają się do nich LCP i NCP dla każdego obsługiwanego protokołu warstwy sieciowej. Domyślna wielkość pola protokołu wynosi 2 bajty, lecz może zostać wynegocjowana redukcja do 1 z użyciem LCP.

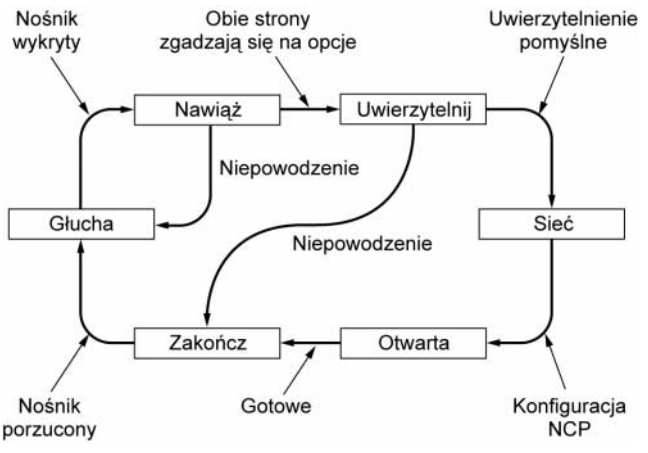
Pole ładunku ma zmienną długość aż do określonego wynegocjowanego maksimum. Jeśli długość nie była negocjowana z użyciem LCP podczas wstępnej konfiguracji linii, używana jest domyślna długość 1500 bajtów. W razie potrzeby po ładunku może następować wypełnienie.

Po polu ładunku następuje pole sumy kontrolnej, które domyślnie ma 2 bajty, lecz można też wynegocjować 4-bajtową sumę kontrolną.

W sumie PPP jest wieloprotokołowym mechanizmem ramkowania nadającym się do użytku w transmisji przez modemy, bitowe łącza HDLC, SONET i inne warstwy fizyczne. Obsługuje wykrywanie błędów, negocjację opcji, kompresję nagłówka i opcjonalnie niezawodną transmisję z użyciem formatu ramki typu HDLC.

Przejdźmy teraz od formatu ramki PPP do sposobu nawiązywania i zrywania połączeń. Schemat (uproszczony) z rysunku 3.21 przedstawia fazy, przez które przechodzi linia podczas włączania, eksploatacji i ponownego wyłączania. Ta sekwencja stosuje się zarówno do połączeń modemowych, jak i pomiędzy routerami.

Protokół zaczyna działanie od linii w stanie GŁUCHA, co oznacza, że nie jest obecny nośnik warstwy fizycznej i że nie istnieje połączenie w warstwie fizycznej. Po nawiązaniu połączenia fizycznego linia przechodzi do NAWIĄŻ. W tym momencie zaczyna się negocjacja opcji LCP, która, jeśli zakończy się powodzeniem, prowadzi do UWIERZYTELNIJ. Teraz obie strony mogą, jeśli chcą, sprawdzić nawzajem swoje tożsamości. Po wejściu do fazy SIEĆ zostaje wywołany odpowiedni protokół NCP do skonfigurowania warstwy sieciowej. Jeśli konfiguracja zostanie zakończona pomyślnie, linia wchodzi w stan OTWARTA i może odbywać się w niej transmisja danych. Po zakończeniu linia przechodzi do fazy ZAKOŃCZ, a z niej do GŁUCHA po porzuceniu nośnika.



RYSUNEK 3.21. Uproszczony schemat faz włączania i wyłączania linii

LCP negocjuje opcje protokołu łącza danych w fazie NAWIĄŻ. Protokół LCP nie zajmuje się w rzeczywistości samymi opcjami, lecz mechanizmem negocjacji. Udostępnia procesowi inicjującemu sposób złożenia propozycji oraz procesowi odpowiadającemu sposób zaakceptowania lub odrzucenia tych propozycji, wszystkich lub wybranych. Oprócz tego umożliwia tym procesom przetestowanie jakości łącza, aby mogły ustalić, czy jest wystarczająca do nawiązania połączenia. Na koniec protokół LCP pozwala też na wyłączanie linii, gdy przestają być potrzebne.

W RFC 1661 zdefiniowanych jest jedenaście typów ramek LFC, które przedstawia tabela 3.1. Cztery typy Configure- pozwalają stronie inicjującej (I) proponować wartości opcji, a stronie odpowiadającej (O) akceptować je lub odrzucać. W drugim przypadku strona odpowiadająca może złożyć alternatywną propozycję lub oświadczyć, że w ogóle nie dopuszcza negocjacji pewnych opcji. Negocjowane opcje i ich proponowane wartości są elementami ramek LCP.

TABELA 3.1.
Typy ramek LCP

Nazwa	Kierunek	Opis
Configure-request	I → O	Lista proponowanych opcji i wartości.
Configure-ack	I ← O	Wszystkie opcje zostały zaakceptowane.
Configure-nak	I ← O	Niektóre opcje nie zostały zaakceptowane.
Configure-reject	I ← O	Niektóre opcje nie podlegają negocjacji.
Terminate-request	I → O	Żądanie wyłączenia linii.
Terminate-ack	I ← O	OK, linia można wyłączyć.
Code-reject	I ← O	Otrzymano nieznane żądanie.
Protocol-reject	I ← O	Zażądano nieznanego protokołu.
Echo-request	I → O	Proszę odesłać tę ramkę z powrotem.
Echo-reply	I ← O	Proszę bardzo, odesłana ramka.
Discard-request	I → O	Proszę zignorować tę ramkę (do testów).

Kody Terminate- wyłączają linię, gdy przestaje być potrzebna. Kody Code-reject i Protocol-reject wskazują, że strona odpowiadająca otrzymała coś, czego nie rozumie. Ta sytuacja może oznaczać, że wystąpił niewykryty błąd transmisji, lecz co bardziej prawdopodobne, obie strony używają innych

wersji protokołu LCP. Typy Echo- służą do testowania jakości linii. Na koniec Discard-request pomaga w znajdowaniu błędów. Jeśli któraś ze stron ma kłopoty z wysyłaniem bitów w kanał, programista może użyć tego typu do testowania. Jeśli ramka zdoła przejść, odbiornik po prostu ją odrzuca, zamiast podejmować jakąś czynność, która mogłaby wprowadzić w błąd osobę testującą połączenie.

Do opcji podlegających negocjacji należą ustawienie maksymalnej wielkości ładunku użytecznego dla ramek danych, włączenie uwierzytelnienia i wybór protokołu do tego celu, włączenie monitorowania linii podczas normalnej pracy oraz wybór różnorodnych opcji kompresji nagłówka.

O protokołach NCP możemy podać niewiele ogólnych wiadomości. Każdy jest przeznaczony dla konkretnego protokołu warstwy sieciowej i pozwala zgłaszać żądania konfiguracji charakterystyczne dla tego protokołu. Na przykład dla IP najważniejszą możliwością jest dynamiczne przydzielanie adresów.

3.7. Podsumowanie

Zadaniem warstwy łącza danych jest konwersja surowego strumienia bitów otrzymanego z warstwy fizycznej na strumień ramek na potrzeby warstwy sieciowej. Stosowane są różne metody ramkowania, w tym zliczanie znaków, napełnianie bajtami i napełnianie bitami. Protokoły łącza danych mogą zapewnić kontrolę błędów do retransmisji uszkodzonych i utraconych ramek. Aby uchronić się przed zalaniem danymi wolnego odbiornika przez szybki nadajnik, protokół łącza danych może też udostępnić sterowanie przepływem. Mechanizm okna przesuwnego jest powszechnie stosowany do wygodnej integracji kontroli błędów i sterowania przepływem.

Protokoły z oknem przesuwnym można podzielić na kategorie według rozmiaru okna nadajnika i odbiornika. Gdy oba są równe 1, protokół jest typu stop-and-wait. Gdy okno nadajnika jest większe niż 1, na przykład aby nadajnik nie blokował się w obwodach z długim czasem propagacji, odbiornik może być zaprogramowany tak, by albo odrzucać wszystkie ramki z wyjątkiem następnej oczekiwanej w sekwencji, albo buforować ramki otrzymane poza kolejnością, dopóki nie staną się potrzebne.

W rozdziale przedstawiliśmy szereg protokołów. Protokół 1. został zaprojektowany dla środowiska wolnego od błędów, w którym odbiornik potrafi obsłużyć każdy przepływ danych. Protokół 2. nadal zakłada środowisko wolne od błędów, lecz wprowadza sterowanie przepływem. Protokół 3. obsługuje błędy przez wprowadzenie numerów sekwencyjnych i użycie algorytmu stop-and-wait. Protokół 4. pozwala na komunikację w obie strony i wprowadza ideę doczepiania potwierdzeń „na barana” (piggybacking). Protokół 5. używa okna przesuwnego z techniką „wróć do n” (go back n). Na koniec protokół 6. używa selektywnych powtórzeń i potwierdzeń negatywnych.

Protokoły mogą być modelowane z użyciem różnych technik pomagających zademonstrować ich poprawność (lub brak takowej). Do tego celu powszechnie stosuje się modele z użyciem automatów skończonych i sieci Petriego.

W wielu sieciach w warstwie łącza danych stosuje się jeden z protokołów o organizacji bitowej — SDLC, HDLC, ADCCP lub LAPB. Wszystkie te protokoły używają bajtów znacznikowych do wytyczania granic ramek i napełniania bitami do zapobiegania przed wystąpieniem bajta znacznikowego w danych. Wszystkie używają też okna przesuwnego do sterowania przepływem. W Internecie protokół PPP jest używany jako podstawowy protokół łącza danych w liniach dwupunktowych.

Zadania

(1) Pakiet z wyższej warstwy jest dzielony na 10 ramek, z których każda ma 80% prawdopodobieństwa dotarcia bez uszkodzeń. Jeśli protokół łącza danych nie przeprowadza żadnej kontroli błędów, ile razy średnio trzeba przesłać wiadomość, aby całość dotarła na miejsce?

(2) W protokole łącza danych używane jest następujące kodowanie znaków: A: 01000111; B: 11100011; FLAG: 01111110; ESC: 11100000. Przedstaw sekwencję bitów wysłanych (binarnie) dla ramki złożonej z czterech znaków: A B ESC FLAG, przy użyciu następujących metod ramkowania:
(a) Zliczanie znaków.
(b) Bajty znacznikowe z napełnianiem bajtami.
(c) Znaczniki początku i końca z napełnianiem bitami.

(3) Poniższy fragment danych występuje w środku strumienia danych, dla którego używany jest algorytm napełniania bajtami opisany w rozdziale: A B ESC C ESC FLAG FLAG D. Jakie będzie wyjście po napełnieniu?

(4) Jeden z kolegów, Sknerus, zwrócił uwagę na to, że oznaczanie końca jednej ramki bajtem znacznikowym i rozpoczynanie następnej drugim bajtem znacznikowym jest marnotrawstwem. Jeden bajt znacznikowy równie dobrze spełni tę rolę, a drugi bajt zostaje nam w kieszeni. Zgadzasz się z tym?

(5) Łańcuch bitów 0111101111101111110 należy przesłać na poziomie warstwy łącza danych. Jak będzie wyglądał ten łańcuch po napełnieniu bitami?

(6) Czy przy stosowaniu napełniania bitami możliwe jest, że utrata, wstawienie lub modyfikacja pojedynczego bitu nie zostanie wykryta przez sumę kontrolną? Jeśli nie, dlaczego? Jeśli tak, w jaki sposób? Czy długość sumy kontrolnej gra tu jakąś rolę?

(7) Czy możesz sobie wyobrazić sytuację, w której protokół z pętlą otwartą (np. kod Hamminga) będzie lepszy od protokołów ze sprzężeniem zwrotnym omawianych w tym rozdziale?

(8) Aby zapewnić większą niezawodność niż ta, jaką daje pojedynczy bit parzystości, schemat kodowania detekcyjnego używa jednego bitu parzystości do kontroli bitów nieparzystych i drugiego dla bitów parzystych. Jaka jest odległość Hamminga dla tego kodu?

(9) Szesnastobitowe komunikaty są przesyłane z użyciem kodu Hamminga. Jak wiele bitów kontrolnych potrzeba, aby zapewnić wykrywanie i korektę przez odbiornik błędów jednobitowych? Pokaż wzorzec bitów przesyłany dla komunikatu 1101001100110101. Załóż, że w kodzie Hamminga używana jest kontrola parzysta.

(10) 8-bitowy bajt o wartości binarnej 10101111 ma być zakodowany z użyciem parzystego kodu Hamminga. Jaka będzie wartość binarna po zakodowaniu?

(11) 12-bitowy kod Hamminga, którego wartość szesnastkowa wynosi `0xE4F`, dociera do odbiornika. Jaka była oryginalna wartość szesnastkowa? Załóż, że najwyżej jeden bit jest błędny.

(12) Jednym ze sposobów wykrywania błędów jest przesyłanie danych jako bloku n wierszy z k bitów na wiersz i dodawanie bitów parzystości do każdego wiersza i każdej kolumny. W prawym dolnym rogu tablicy znajduje się bit parzystości kontrolujący swój wiersz i kolumnę. Czy taki schemat pozwala wykryć wszystkie pojedyncze błędy? Podwójne? Potrójne?

(13) Blok bitów o rozmiarach n wierszy i k kolumn używa pionowych i poziomych bitów parzystości do wykrywania błędów. Załóżmy, że dokładnie 4 bity zostały zanegowane z powodu błędów transmisji. Wyprowadź wzór na prawdopodobieństwo, że błąd nie zostanie wykryty.

(14) Jaka będzie reszta z dzielenia wielomianu $x^7 + x^5 + x$ przez wielomian generujący $x^3 + 1$?

(15) Strumień bitów 10011101 jest przesyłany z użyciem standardowej metody CRC opisanej w rozdziale. Wielomian generujący to $x^3 + 1$. Przedstaw faktyczny przesłany łańcuch bitów. Załóżmy, że trzeci bit od lewej został zanegowany podczas transmisji. Pokaż, że ten błąd zostanie wykryty po stronie odbiornika.

(16) Protokoły łącza danych niemal zawsze umieszczają CRC w stopce, a nie w nagłówku. Dlaczego?
(17) Kanał ma przepustowość 4 kb/s i opóźnienie propagacji równe 20 ms. Dla jakiego zakresu wielkości ramek protokół stop-and-wait zapewni wydajność równą przynajmniej 50%?
(18) Łącze T1 o długości 3000 km służy do przesyłania 64-bajtowych ramek z użyciem protokołu 5. Jeśli opóźnienie propagacji wynosi 6μs/km, to ile bitów powinien mieć numer sekwencyjny?
(19) Czy w protokole 3. możliwe jest, że nadajnik uruchomi czasomierz, gdy ten będzie już uruchomiony? Jeśli tak, jak to może się stać? Jeśli nie, dlaczego jest to niemożliwe?
(20) Wyobraź sobie protokół z oknem przesuwnym używający tak wielu bitów na numer sekwencyjny, że zawinięcie nigdy nie wystąpi. Jakie zależności muszą być spełnione pomiędzy czterema krawędziami okien i rozmiarem okna, który jest stały i taki sam dla nadawcy i odbiorcy?
(21) Gdyby procedura `between` w protokole 5. sprawdzała warunek $a \leq b \leq c$ zamiast $a \leq b < c$, czy miałoby to jakiś wpływ na poprawność lub wydajność protokołu? Uzasadnij swoją odpowiedź.
(22) Gdy w protokole 6. przychodzi ramka danych, dokonywana jest kontrola, czy numer sekwencyjny różni się od spodziewanego, i czy `no_nak` jest prawdą. Jeśli oba warunki są spełnione, zostaje wysłana ramka NAK; w przeciwnym razie zostaje uruchomiony czasomierz pomocniczy. Załóżmy, że został pominięty warunek `else`. Czy zmieni to poprawność protokołu?
(23) Załóżmy, że usuniemy z kodu trzyinstrukcyjną pętlę `while` w okolicach końca protokołu 6. Czy wpłynie to na poprawność protokołu czy tylko na wydajność? Uzasadnij odpowiedź.
(24) Załóżmy, że z instrukcji `switch` w protokole 6. został usunięty przypadek dla błędów sumy kontrolnej. Jak zmieni to działanie protokołu?
(25) W protokole 6. kod dla `frame_arrival` zawiera sekcję używaną dla NAK. Ta sekcja jest wywoływana, gdy przychodząca ramka jest NAK i jest spełniony inny warunek. Przedstaw scenariusz, w którym obecność tego drugiego warunku jest niezbędna.
(26) Wyobraź sobie, że piszesz oprogramowanie warstwy łącza danych dla linii używanej do przesyłania danych do siebie, lecz nie od siebie. Drugi koniec stosuje HDLC z 3-bitowym numerem sekwencyjnym i oknem o rozmiarze siedmiu ramek. Chcesz buforować jak najwięcej ramek niezgodnych z sekwencją, jak to tylko możliwe, aby zwiększyć wydajność, lecz możesz zmodyfikować tylko oprogramowanie po stronie nadajnika. Czy możliwe jest użycie okna odbioru większego niż 1 i jednoczesne zagwarantowanie, że protokół nigdy nie zawiedzie? Jeśli tak, jakie jest największe okno, które można użyć bezpiecznie?
(27) Rozważmy działanie protokołu 6. w łączu 1 Mb/s wolnym od błędów. Maksymalna wielkość ramki wynosi 1000 bitów. Nowe pakiety są generowane co sekundę. Dopuszczalny interwał oczekiwania wynosi 10 ms. Gdyby został wyeliminowany specjalny czasomierz potwierdzeń, występowałyby niepotrzebne przekroczenia czasu oczekiwania. Średnio ile razy zostałby wysłany każdy komunikat?
(28) W protokole 6. `MAX_SEQ` $= 2^n - 1$. Wprawdzie ten warunek jest ewidentnie pożądany, aby najskuteczniej wykorzystać bity nagłówka, to nie zademonstrowaliśmy, że jest konieczny. Czy protokół będzie funkcjonował prawidłowo dla, na przykład, `MAX_SEQ` = 4?
(29) Ramki o długości 1000 bitów są wysyłane kanałem 1 Mb/s z użyciem satelity geostacjonarnego, dla którego czas propagacji od Ziemi wynosi 270 ms. Potwierdzenia zawsze są transmitowane „na barana" w ramkach danych. Nagłówki są bardzo krótkie. Używane są trzybitowe numery sekwencyjne. Jakie jest maksymalne wykorzystanie kanału osiągalne przy użyciu:
(a) protokołu stop-and-wait,
(b) protokołu 5.,
(c) protokołu 6.?
(30) Oblicz, jaka część pasma jest marnowana na dodatkowe informacje (nagłówki i retransmisje) w protokole 6. w mocno obciążonym kanale satelitarnym 50 kb/s z ramkami danych złożonymi z 40 bitów nagłówka i 3960 bitów danych. Załóż, że czas propagacji sygnału z Ziemi do satelity wynosi 270 ms. Ramki ACK nigdy nie występują. Ramki NAK mają długość 40 bitów. Stopa błędów dla ramek danych wynosi 1%, a dla ramek NAK jest pomijalna. Numery sekwencyjne mają 8 bitów.

(31) Rozważmy wolny od błędów kanał satelitarny 64 kb/s, służący do wysyłania 512-bajtowych ramek danych w jednym kierunku, z bardzo krótkimi potwierdzeniami przesyłanymi w drugą stronę. Jaka jest maksymalna zdolność przepustowa kanału dla okien o rozmiarach 1, 7, 15 i 127? Czas propagacji Ziemia-satelita wynosi 270 ms.

(32) Kabel o długości 100 km przesyła dane z przepływnością T1. Prędkość propagacji sygnału w kablu wynosi 2/3 prędkości światła w próżni. Ile bitów mieści się w kablu?

(33) Załóżmy, że zamodelujemy protokół 4. z użyciem modelu automatów skończonych. Ile stanów istnieje dla każdego automatu? Ile stanów istnieje dla kanału komunikacyjnego? Ile stanów istnieje dla kompletnego systemu (dwa automaty i kanał)? Zignoruj błędy sumy kontrolnej.

(34) Podaj sekwencję odpaleń dla sieci Petriego z rysunku 3.16 odpowiadającą sekwencji stanów (000), (01A), (01–), (010), (01a) na rysunku 3.14. Objaśnij słownie, co reprezentuje ta sekwencja.

(35) Mając reguły przejść $AC \rightarrow B$, $B \rightarrow AC$, $CD \rightarrow E$ i $E \rightarrow CD$, narysuj opisaną sieć Petriego. Na podstawie tej sieci narysuj graf stanów skończonych osiągalny ze stanu początkowego *ACD*. Jaką dobrze znaną koncepcję modelują te reguły przejść?

(36) PPP opiera się w dużym stopniu na HDLC, który używa napełniania bitami do zapobiegania niejasnościom, powodowanym przez przypadkowe wystąpienia bajta znacznikowego w ładunku ramki. Podaj przynajmniej jeden powód, dla którego PPP używa zamiast tego napełniania bajtami.

(37) Jaki jest minimalny narzut danych przy wysyłaniu pakietu IP przez PPP? Wlicz tylko dane wprowadzone przez PPP, pomijając nagłówek IP.

(38) Celem tego ćwiczenia laboratoryjnego jest zaimplementowanie mechanizmu wykrywania błędów z użyciem standardowego algorytmu CRC opisanego w rozdziale. Napisz dwa programy, generator i weryfikator. Generator wczytuje z wejścia standardowego *n*-bitowy komunikat jako łańcuch zer i jedynek zapisanych tekstem ASCII. W drugim wierszu znajduje się *k*-bitowy wielomian, również w ASCII. Powinien wysłać na wyjście standardowe wiersz tekstu ASCII złożony z $n + k$ zer i jedynek reprezentujący komunikat do wysłania. Następnie wysyła wielomian tak, jak go wczytał. Program weryfikujący wczytuje wyjście programu generującego i wysyła na wyjście komunikat informujący, czy dane są poprawne czy nie. Na koniec napisz program modyfikujący *zmieniacz*, który neguje jeden bit z pierwszego wiersza, zależny od argumentu (bity liczone są od lewej; pierwszy bit po lewej ma numer 1), lecz kopiuje poprawnie całą resztę obu wierszy. Wpisanie:

```
generator <plik | weryfikator
```

powinno dać komunikat o poprawności danych, lecz wpisanie:

```
generator <plik | zmieniacz argument | weryfikator
```

powinno dać komunikat o błędzie.

(39) Napisz program symulujący zachowanie sieci Petriego. Program powinien wczytywać reguły przejść i listę stanów odpowiadające wydawaniu przez warstwę sieciową nowego pakietu lub przyjmowanie nowego pakietu. Ze stanu początkowego, również wczytanego, program powinien wybrać dowolnie włączone przejścia i odpalić je, sprawdzając, czy host kiedykolwiek może przyjąć 2 pakiety bez nadania w tym samym czasie przez drugiego hosta nowego pakietu.