Uniwersytet Warszawski
Wydział Matematyki, Informatyki i Mechaniki

Jerzy Ziemiański
Nr albumu: 181074

Protokół niezawodnego rozgłaszania w sieci lokalnej

Praca magisterska
na kierunku INFORMATYKA

Praca wykonana pod kierunkiem
dr Janiny Mincer-Daszkiewicz
Instytut Informatyki

Wrzesień 2004

Pracę przedkładam do oceny

Data Podpis autora pracy:

Praca jest gotowa do oceny przez recenzenta

Data Podpis kierującego pracą:

Streszczenie

W pracy przedstawiono istniejące protokoły niezawodnego rozgłaszania, ich zalety oraz wady. Zaproponowano nowy – przeznaczony do wykorzystania w sieciach lokalnych i charakteryzujący się kilkoma ciekawymi cechami. Umożliwia między innymi sprawne rozsyłanie krótkich strumieni danych, definiowanie wymaganego poziomu niezawodności indywidualnie dla każdego strumienia i klienta oraz wprowadza tryb „podsłuchiwania", w którym klient ma gwarancję otrzymania strumieni danych we właściwej kolejności. Testy prototypu implementacji dowiodły, że protokół jest wydajny i może z powodzeniem zostać wykorzystany w praktyce.

Słowa kluczowe

protokół niezawodnego rozgłaszania, rozsyłanie grupowe, sieć lokalna, RFC 2887, protokół z potwierdzeniami, Java

Klasyfikacja tematyczna

C.2.2 [Computer-Communication Networks]: Network Protocols

Spis treści

Spis tabel

Rozdział 1

Wprowadzenie

W sieciach lokalnych często występuje konieczność posłużenia się tą samą informacją na wielu komputerach. Wiąże się to z koniecznością jej niezawodnego dostarczenia. Zwykle rozwiązywane jest to przez przesłanie tej informacji do każdego komputera z osobna, co jest oczywiście nieefektywne.

Problem jest często lekceważony, a wytłumaczeniem takiej postawy jest argument, że przepustowość sieci jest wystarczająco duża. W wielu przypadkach jest to zrozumiałe, gdyż spotykane są już sieci o przepustowości nawet 1Gbit/s, jednak nadal zdarzają się miejsca, gdzie faktyczna przepustowość jest o kilka rzędów wielkości mniejsza i stanowi ona tzw. wąskie gardło. Ponadto w jednej sieci lokalnej może działać wiele systemów, z których każdy wymaga pewnej przepustowości. W coraz większym stopniu wykorzystywane są narzędzia do pracy zdalnej, takie jak VNC (Virtual Network Computing) czy Citrix MetaFrame™, które potrafią w znacznym stopniu obciążyć sieć lokalną.

Sprzęt oraz biblioteki systemu operacyjnego już od dawna dostarczają narzędzi umożliwiających wysłanie tej samej informacji do wielu komputerów jednocześnie poprzez rozgłaszanie (ang. *broadcast*) oraz rozsyłanie grupowe (ang. *multicast*). Niestety nie jest zapewniona niezawodność, co sprawia, iż ich wykorzystanie jest marginalne – służą one w większości przypadków jedynie do „odnajdywania się" w sieci.

W niniejszej pracy chciałem przedstawić możliwe rozwiązania postawionego problemu, czyli protokoły zapewniające niezawodne rozgłaszanie. Ponadto zaproponowałem własny protokół, interfejs programisty do jego obsługi (API), jego implementację oraz przykład wykorzystania w praktyce. Trudność zadania zależy od przyjętych założeń. Najważniejszym uproszczeniem było przyjęcie, że protokół ma działać w sieci lokalnej. Pozostałe założenia, takie jak ograniczenie wykorzystywanej pamięci, zapewnienie kolejności dostarczania danych, równoległe przesyłanie wielu strumieni danych, zapewnienie dużej wydajności przy przesyłaniu krótkich strumieni, możliwość dynamicznego definiowania stopnia niezawodności na poziomie klienta i strumienia danych, czy możliwość implementacji w Javie, sprawiają, że zadania nie można nazwać banalnym.

Praca składa się z ośmiu rozdziałów. W rozdziale 2 przedstawiam istniejące protokoły oraz ich wady i zalety. W rozdziale 3 opisuję wymagania, jakie chciałbym, aby moje rozwiązanie spełniało. Problemy, z jakimi się zetknąłem oraz decyzje, jakie podjąłem zawarte są w rozdziale 4. Opis zaprojektowanego przeze mnie protokołu znajduje się w rozdziale 5. Rozdział 6 zawiera szczegóły dotyczące implementacji, a w rozdziale 7 podaję przykład zastosowania. Analiza przeprowadzonych testów znajduje się w rozdziale 8, a propozycje dalszych prac w rozdziale 9.

Przed rozpoczęciem lektury warto zapoznać się ze skróconym opisem pojęć, którymi po-

sługuję się w całej pracy – słowniczek znajduje się w dodatku A. W dodatku B zamieszczono opis zawartości płyty CD, dołączonej do pracy.

Rozdział 2

Istniejące protokoły

Protokołami niezawodnego rozgłaszania zajmowano się od dawna. Istnieje wiele gotowych rozwiązań, które są wykorzystywane w specyficznych sytuacjach. Wynikiem wieloletnich badań i prób stworzenia takich protokołów jest m.in. stwierdzenie, że nie jest możliwe stworzenie jednego uniwersalnego protokołu niezawodnego rozgłaszania, który byłby optymalny w każdych warunkach. Udało się jednak wydzielić części składowe takich protokołów ([RFC3048]), dzięki czemu poprawiając jedną z nich można się przyczynić do poprawy działania całej rodziny protokołów. Ponadto można tworzyć zupełnie nowe protokoły składając je wedle potrzeb z gotowych elementów.

Jako argument potwierdzający tezę o niemożności stworzenia uniwersalnego protokołu przytoczę dwa zastosowania – nadawanie programów telewizyjnych w internecie oraz synchronizacja plików w sieci lokalnej. Różnice są przedstawione w tabeli 2.1.

W obu przedstawionych przypadkach mamy do czynienia z zastosowaniami protokołów, które kryją się pod wspólną nazwą protokołów niezawodnego rozgłaszania. Wymagania stawiane tym protokołom są jednak zasadniczo różne, co sprawia, że zaprojektowanie jednego, wspólnego protokołu, który spełniałby je wszystkie w zadowalający sposób, jeśli nie jest niemożliwe, to jest bardzo trudne.

2.1. Podział ze względu na sposób zapewnienia niezawodności

Istnieją trzy główne sposoby zapewnienia niezawodności:

- FEC – przesyłanie nadmiarowych informacji umożliwiających odtworzenie części zgubionych danych;
- ACK – przesyłanie przez klientów potwierdzeń otrzymania danych;
- NAK – przesyłanie przez klientów żądań retransmisji w przypadku stwierdzenia zgubienia danych.

Oczywiście nic nie stoi na przeszkodzie, aby te metody ze sobą połączyć.

2.2. Protokoły FEC

Skrót FEC wywodzi się od angielskiego „Forward Error Correction”, czyli korekcja błędów z wyprzedzeniem, co chyba dobrze oddaje ideę tej grupy protokołów. Do przesyłanych pakietów dołączane są redundantne dane mające na celu umożliwienie rekonstrukcji zagubionych pakietów.

	Nadawanie programów telewizyjnych w internecie	Sychronizacja plików w sieci lokalnej
Charakterystyka przesyłanych danych	Jeden lub kilka strumieni (o różnych prędkościach) bez wyraźnie określonego początku i końca.	Wiele bloków lub strumieni o rozmaitej, ale znanej długości.
Wymagania dotyczące niezawodności	Zgubienie części danych zwykle powoduje jedynie utratę jakości odtwarzanego dźwięku i obrazu.	Zgubienie choćby jednego bitu jest niedopuszczalne.
Wymagania dotyczące skalowalności	Bardzo duże. Zwykle stosuje się serwery pośredniczące w celu jej zwiększenia.	Małe. Zwykle poniżej kilkudziesięciu klientów.
Odporność na zmiany w prędkości przesyłania danych	Mała. Klient przetwarza dane w sposób ciągły – nie można przesłać zbyt dużej ilości danych z wyprzedzeniem, ponieważ ma on ograniczone możliwości ich przechowywania, a nie może ich wykorzystać od razu. Sytuacja odwrotna, czyli spowolnienie, bądź przerwy w przekazywaniu danych, też nie jest dobra.	Duża. Najważniejszy jest fakt przesłania całego pliku – im szybciej, tym lepiej.
Opóźnienie przesyłania danych	Powinno być stałe.	Powinno być jak najmniejsze.
Wpływ klienta na zachowanie serwera	Należy go zminimalizować, gdyż utrudniałoby to skalowalność. W szczególności klient nie może wymuszać wstrzymania przesyłanych danych.	Duży. Może wymusić retransmisję oraz wstrzymanie przesyłania nowych danych.
Przykładowe sposoby zapewnienia niezawodności (pełnej lub częściowej)	Wysyłanie nadmiarowych danych (ang. *forward error correction*). Retransmisje zwykle nie wchodzą w grę ze względu na konieczność utrzymania stałego opóźnienia.	Potwierdzanie otrzymanych danych, żądania retransmisji.

Tabela 2.1: Porównanie wymagań stawianych protokołom niezawodnego rozgłaszania

Niestety protokoły wykorzystujące jedynie FEC nie zapewniają pełnej niezawodności, co wyklucza ich zastosowanie w rozwiązaniu postawionego przeze mnie problemu. Są one jednak powszechnie wykorzystywane do nadawania strumieni dźwięków i obrazów w sieci Internet.

W niniejszej pracy ograniczę się do przedstawienia najprostszego mechanizmu korekcji błędów. Opisy oraz odnośniki do pozostałych, bardziej zaawansowanych, można znaleźć w [RFC3453].

W najprostszym protokole FEC przesyłane są dodatkowe pakiety zawierające, poza nagłówkami, jedynie sumy kontrolne danych z innych pakietów. Umożliwia to rekonstrukcję zgubionych danych.

Załóżmy, że wszystkie wysyłane pakiety mają ustalony rozmiar oraz, że na każde N wysłanych pakietów co najwyżej jeden jest gubiony. Wówczas wystarczy po wysłaniu każdej grupy N-1 pakietów wysłać pakiet, zawierający wynik funkcji XOR zastosowanej na danych z poprzednich pakietów. Dane z dowolnego jednego zgubionego pakietu można odtworzyć obliczając funkcję XOR na danych z pozostałych. Zauważmy, że najprostszy przypadek, dla N=2, sprowadza się do wysyłania każdego pakietu dwukrotnie.

Rozwiązanie to ma niestety wady:

- zakłada, że pakiety mają stały rozmiar,
- serwer musi obliczać sumy XOR wszystkich wysyłanych pakietów,
- zapewnia odporność tylko na zgubienie pojedynczych pakietów – brak pełnej niezawodności,
- w przypadku zgubienia pierwszego pakietu z grupy klient musi czekać aż odbierze wszystkie pozostałe, aby móc go zrekonstruować,
- rozmiar bufora klienta zależy od wymaganego poziomu niezawodności (jest proporcjonalny do N oraz rozmiaru pakietu),
- zawsze wysyłane są dodatkowe dane, niezależnie od tego czy jest taka potrzeba.

2.3. Protokoły ACK

Protokoły z potwierdzeniami pozytywnymi charakteryzują się tym, że klienci potwierdzają otrzymanie danych.

Wady:

— Klienci generują dodatkowy ruch w sieci nawet wtedy, gdy wszystkie dane docierają poprawnie – tym większy, im więcej jest klientów.

— Słaba skalowalność uniemożliwiająca zastosowanie w dużych sieciach.

Zalety:

\+ Jako jedyne zapewniają pełną niezawodność przy minimalnym zapotrzebowaniu na pamięć.

Istniejące protokoły różnią się sposobem potwierdzania.

2.3.1. Protokół podstawowy

Protokół podstawowy to taki, w którym każdy pakiet otrzymany przez klienta jest przez niego od razu potwierdzany. W przypadku braku otrzymania potwierdzenia przez serwer przez pewien czas pakiet jest ponownie rozgłaszany. Serwer musi pamiętać listę klientów wraz z numerami pakietów przez nich potwierdzonymi oraz czas, jaki upłynął od wysłania pakietów.

W porównaniu z indywidualnymi połączeniami (rezygnacją z rozgłaszania) można wskazać następujące zalety:

- Zmniejszenie zużycia zasobów systemowych serwera (wystarczy jedno gniazdo do rozgłaszania oraz jedno gniazdo do odbierania potwierdzeń);
- Do dwóch razy mniejsze obciążenie sieci.

2.3.2. Protokół grupujący potwierdzenia

Protokół grupujący potwierdzenia jest drobną modyfikacją protokołu podstawowego polegającą na grupowaniu przez klienta potwierdzeń. Klient zwleka przez pewien czas z potwierdzaniem pakietów po to, aby zmniejszyć ruch w sieci. Skalowalność jest kilka razy większa niż w przypadku protokołu podstawowego, jednak nadal nie nadaje się on do zastosowania w sieci Internet.

2.3.3. Protokół drzewiasty

Protokół drzewiasty (ang. *tree-based protocol*) ma na celu dalsze zmniejszenie liczby potwierdzeń i umożliwienie wykorzystania go w sieci Internet. Jest to możliwe dzięki wprowadzeniu wyróżnionych węzłów pełniących podwójną funkcję – serwera i klienta. Tworzą one hierarchię – drzewo, którego korzeniem jest serwer, a klienci liśćmi.

2.4. Protokoły NAK

Protokoły z potwierdzeniami negatywnymi to takie, w których serwer zakłada, że klienci poprawnie otrzymali dane. Klient sam musi stwierdzić, że nie otrzymał wszystkich danych i wysłać prośbę o retransmisję.

Największą ich zaletą jest znikomy wpływ liczby klientów na obciążenie serwera w optymalnych warunkach, czyli takich, gdzie pakiety nie są gubione (niezależnie od powodu). W skrajnych przypadkach serwer w ogóle nic nie wie o klientach i ich liczbie. Reaguje jedynie na prośby o retransmisje.

Początkowo chciałem zastosować taki protokół do replikacji danych, jednak ma on bardzo poważne wady ujawniające się wraz z pogorszeniem warunków, w których ma działać. Okazuje się, że protokoły NAK mogą mieć bardzo duże wymagania pamięciowe. Zależą one od wymaganego poziomu niezawodności, jakości sieci oraz od maksymalnego czasu reakcji klienta i serwera (maksymalny czas potrzeby do przetworzenia pakietu, czyli od jego otrzymania do wysłania odpowiedzi lub podjęcia innej czynności). W sieci lokalnej można pominąć kwestię jej jakości – pozostaje wymagana niezawodność i maksymalny czas reakcji klienta. Wymagania pamięciowe serwera M_s można scharakteryzować następującym wzorem:

$$M_s = B(2T_p + T_{rk} + T_{rs}),$$

gdzie T_p – czas przesłania pakietu przez sieć, T_{rk} – czas reakcji klienta, T_{rs} – czas reakcji serwera, B – przepustowość sieci.

Można przyjąć, że dla danej sieci B oraz T_p są ustalone. Jedynymi niewiadomymi są T_{rk} oraz T_{rs}. Gdyby przyjąć, że protokół jest zaimplementowany jako część jądra systemu operacyjnego, to można by te wielkości oszacować z góry – można założyć, że w większości przypadków byłyby mniejsze od $10ms$. Wobec tego dla sieci o przepustowośći $10Mb/s$ i czasu przesłania pakietu równym $5ms$ mamy:

$$M_s = 10Mb/s \times (2 \times 5ms + 10ms + 10ms) = 300Kb \approx 37KB$$

Widać, że jest to wartość akceptowalna. Niestety gdybyśmy chcieli zaimplementować taki protokół jako zwykły proces, to okazuje się, że czas reakcji może być dużo dłuższy. Główne powody to wpływ innych procesów oraz zastosowanie pamięci wirtualnej (istnieje niebezpieczeństwo, że proces zostanie usunięty z pamięci operacyjnej i konieczne będzie wczytanie go z pliku wymiany, co związane jest ze sporym opóźnieniem). Sytuacja wygląda jeszcze gorzej w przypadku zastosowania maszyny wirtualnej, gdzie same przestoje związane z pracą odśmiecarki pamięci mogą w skrajnych przypadkach przekraczać kilka sekund.

Policzmy ile pamięci potrzeba do zapewnienia niezawodności, przy założeniu, że maksymalny czas reakcji wynosi $5s$, a parametry sieci są takie jak poprzednio:

$$M_s = 10Mb/s \times (2 \times 5ms + 5s + 5s) > 100Mb \approx 12MB$$

Nadal nie jest to przypadek najgorszy, gdyż sieć może być jeszcze szybsza, a przestoje dłuższe:

$$M_s = 100Mb/s \times (2 \times 5ms + 5s + 5s) > 1000Mb \approx 125MB$$

Gdybyśmy mieli gwarancję co do maksymalnego czasu przestoju oraz maksymalnej przepustowości sieci, to moglibyśmy w pewnych warunkach rozważać zastosowanie takiego protokołu. Uznałem jednak, że koszt pamięciowy jest zbyt duży, a potrzeba niezawodności zbyt ważna, abym dopuszczał jego zastosowanie.

2.4.1. Ankietowanie

Problem zbyt dużych wymagań pamięciowych można rozwiązać poprzez ankietowanie (ang. *polling*) co jakiś czas klientów. Dane z bufora serwera zwalniane są dopiero po otrzymaniu odpowiedzi od wszystkich klientów, że otrzymali dane.

Największym wyzwaniem przy projektowaniu takich protokołów jest dobranie odpowiedniej strategii rozsyłania ankiet.

Rozsyłanie zbyt rzadkie albo dopiero po stwierdzeniu takiej konieczności (po stwierdzeniu wyczerpania wolnego miejsca w buforze) może doprowadzić do niepotrzebnych przestojów przy wysyłaniu. Serwer nie może zwolnić pamięci dopóty, dopóki nie otrzyma odpowiedzi na wysłane ankiety. Rozsyłanie zbyt częste upodabnia protokół NAK do protokołu ACK, również pod względem wad. W pracy [Lane] zawarto wniosek, że w sieci lokalnej najlepiej sprawdza się wysyłanie ankiet co 80% – 90% rozmiaru bufora.

Kierowanie ankiet do wszystkich klientów na raz może spowodować powstawanie konfliktów przy próbie wysłania w tym samym czasie wszystkich odpowiedzi. Nie jest to groźne jedynie w sieciach o topologii gwiazdy, gdzie dane wysyłane przez jednego klienta nie trafiają do pozostałych.

Rozdział 3

Wymagania

W rozdziale tym przedstawione są wymagania dotyczące protokołu.

3.1. Protokół

W sekcji tej zdefiniowane zostały wymagania zgodnie z [RFC2887]. Wspomniany dokument przedstawia pytania, na jakie należy odpowiedzieć przy projektowaniu protokołu niezawodnego rozgłaszania, możliwe odpowiedzi, czyli rozwiązania, oraz skutki ich wyboru.

3.1.1. Sieć

Protokół jest przewidziany do pracy w sieci lokalnej, czyli takiej, która:

- jest administrowana przez jedną jednostkę organizacyjną,
- jest zaufana (istnieje możliwość wykluczenia osób zakłócających pracę protokołu),
- dostarcza sprzętowych mechanizmów rozgłaszania,
- topologia jest ustalona (nie zmienia się często w czasie),
- czas przesłania pakietu jest niewielki i przewidywalny (rzędu milisekund).

Nie przewiduję wykorzystania protokołu w Internecie.

3.1.2. Skalowalność

Wymagania wynikają z przyjęcia sieci lokalnej jako miejsca zastosowania. Wystarczy, że protokół będzie efektywny przy kilkudziesięciu klientach. Dzięki temu można sobie pozwolić na potwierdzanie bezpośrednio przez klientów otrzymanych danych rezygnując ze skomplikowanych mechanizmów jakich mają zastosowanie w protokołach *drzewiastych*.

3.1.3. Czy wszyscy zainteresowani otrzymali dane?

Najpierw trzeba zdecydować co oznacza, że wszyscy zainteresowani otrzymali dane. Otrzymaniem danych może być zapisanie ich w buforze systemu operacyjnego, dodatkowym buforze protokołu, o ile taki istnieje, dostarczenie aplikacji klienckiej lub też wykorzystanie tej informacji przez aplikację kliencką na przykład poprzez zapisanie jej na dysku twardym bądź

w bazie danych. Jak widać definicja bardzo mocno zależy od zastosowania protokołu. Przyjąłem, że wystarczającym wymaganiem jest dostarczenie informacji, że dane zostały odczytane przez klienta (implementację protokołu) z bufora systemu operacyjnego. Potwierdzenie otrzymania danych w pozostałych wymienionych znaczeniach można zrealizować na innym poziomie, na przykład poprzez przesyłanie potwierdzeń przez połączenia TCP/IP zestawione pomiędzy serwerem a każdym klientem osobno.

3.1.4. Pełna lub częściowa niezawodność

Nie wszystkie aplikacje wymagają pełnej niezawodności. Przykładem może być przesyłanie dźwięku lub obrazu, gdzie zgubiony pakiet wpływa jedynie na pogorszenie jakości, a nie na całkowitą utratę informacji. Chciałbym aby ten protokół można było wykorzystać do przesyłania danych bez ograniczania ich charakteru, w związku z czym musi być w pełni niezawodny. Chciałbym również, aby była możliwość zrezygnowania z pełnej niezawodności w celu zwiększenia wydajności, przy założeniu jednak, że klient będzie wiedział, że dane są niekompletne.

3.1.5. Wyróżnianie klientów

Klienci mogą się różnić wymaganym poziomem niezawodności. Jeden klient może sobie zażyczyć pełnej niezawodności dostarczenia danych, podczas, gdy inny może jej nie wymagać. Przykładem może być przesyłanie danych na życzenie. Klient A zażyczył sobie pewnych danych. Klientowi B nie są one w tej chwili potrzebne, ale za jakiś czas mogą się przydać, wobec czego je odbiera. Jeśli klient B nie otrzyma wszystkich danych, to trudno – poprosi o nie wówczas, gdy będą mu potrzebne.

Wierzę, że takie podejście umożliwi zwiększenie efektywności protokołu, gdyż zminimalizuje negatywny wpływ „słabych ogniw" na całą resztę.

3.1.6. Ograniczenia czasowe dostarczenia danych

Protokół powinien dostarczać dane tak szybko, jak to jest możliwe. Brak możliwości dostarczenia danych w określonym czasie powinien skutkować „odłączeniem" problematycznego klienta z możliwością uzyskania informacji o tym fakcie przez serwer. Wystarczy, że czas ten będzie konfigurowalny dla połączenia. Konfigurowalność na poziomie klienta czy pakietu danych nie jest wymagana.

3.1.7. Czy dostarczone dane muszą być uporządkowane

Protokół musi gwarantować dostarczenie klientowi danych w obrębie kanału w takiej kolejności, w jakiej zostały wysłane.

3.1.8. Obsługa wielu serwerów

Musi być możliwość poprawnego działania wielu serwerów w jednej sieci lokalnej jednocześnie, pod warunkiem, że będą w pełni niezależne, czyli wykorzystywać będą różne gniazda (ang. *sockets*), czyli pary (adres, port). Protokół zakłada topologię 1-N, czyli jeden serwer, wielu klientów.

3.1.9. Bezpieczeństwo

W niniejszej pracy nie zajmuję się kwestią bezpieczeństwa, głównie dlatego, że protokół ma być wykorzystany w sieci lokalnej, czyli zaufanej. Nie przewiduję zastosowania go do przesyłania danych poufnych, chociaż wierzę, że jest to możliwe za pomocą szyfrowania strumienia danych.

3.1.10. Pozostałe wymagania

1. Brak ograniczeń na charakter przesyłanych danych. Nie można zabronić wykorzystania protokołu do przesyłania nieskończonych strumieni danych. Algorytm powinien się zachowywać efektywnie w przypadku przesyłania dużej liczby krótkich strumieni, czyli małych pakietów.

2. Możliwość wysyłania wielu strumieni danych równolegle. Na przykład przy przesyłaniu plików pierwszy klient może sobie zamówić bardzo duży plik, drugi bardzo mały. Nie chcemy aby drugi odbiorca czekał na przesłanie pliku skierowanego do pierwszego odbiorcy.

3. Możliwość dynamicznego ustalania, które dane który klient musi otrzymać oraz które dane dla którego klienta nie są istotne. Serwer nie może być spowalniany przez klientów, którzy nie są zainteresowani przesyłanymi właśnie danymi.

4. Rozsądne (ograniczone) wymagania pamięciowe. Nie można zakładać, że pamięć jest nieskończenie duża.

5. Rozsądny narzut na ilość przesyłanych danych.

6. Poprawne działanie w niesprzyjających warunkach, takich jak występowanie przerw w dostarczaniu pakietów, przestoje serwera lub klienta w związku na przykład z pracą odśmiecarki maszyny wirtualnej. Nie może wówczas dochodzić do zrywania transmisji. Musi być możliwość zaimplementowania algorytmu jako części aplikacji, a nie jako sterownika lub modułu jądra systemu operacyjnego.

7. Możliwość pracy wielu niezależnych systemów wykorzystujących ten protokół w jednej sieci lokalnej bez odczuwalnego wpływu na wydajność któregokolwiek (poza oczywiście fizycznymi ograniczeniami, takimi jak przepustowość sieci czy wydajność komputerów).

3.2. Interfejs programistyczny (API)

1. Prosty. Nie może być skomplikowany, bo nikt go nie będzie w stanie użyć.

2. Odporny na błędy. Nie można zrzucać na użytkownika (klienta API) zbyt dużej odpowiedzialności. W przypadku błędów w sposobie używania należy w miarę możliwości podpowiadać prawidłowe użycie. Jedyne wymagania stawiane klientowi to konieczność zamykania otwartych połączeń oraz prawidłowa obsługa wątków.

3. Zbliżony do istniejących. Jeśli istnieje taka możliwość, to lepiej uzupełnić istniejący interfejs o elementy charakterystyczne dla tego protokołu niż tworzyć zupełnie nowy. Dzięki temu nauczenie się go i zastosowanie będzie łatwiejsze.

3.3. Implementacja

1. Przenośna. Musi być możliwość uruchomienia na różnych systemach operacyjnych – przynajmniej Windows i Linux.

2. Wydajna. Aplikacja ją wykorzystująca nie może być wolniejsza niż działająca z najprostszym protokołem (wysyłanie tych samych danych do każdego klienta osobno).

3. Java. Postawiłem sobie za cel zaimplementowanie tego protokołu w Javie. Jest to o tyle trudne, że odśmiecarka pamięci może spowodować bardzo długie, nawet kilkusekundowe, przerwy w działaniu aplikacji.

Zdecydowałem się na implementację w Javie z dwóch powodów. Po pierwsze widzę zastosowania w aplikacjach, które już zostały napisane w Javie. Wykorzystanie innego języka programowania utrudniłoby integrację z tymi aplikacjami oraz wymagałoby więcej pracy w celu zapewnienia przenośności między różnymi systemami operacyjnymi. Po drugie chciałem sprawdzić czy Java nadaje się do takiego zastosowania oraz na ile utrudnia lub ułatwia to zadanie.

Rozdział 4

Problemy

W rozdziale tym opisuję problemy z jakimi się zetknąłem oraz decyzje jakie musiałem podjąć. Część jest związana z API, część z samym protokołem, a część z implementacją. Taka też jest kolejność ich prezentacji.

4.1. Pakiety czy strumienie

Pierwszą decyzją jaką musiałem podjąć to czy przesyłane dane należy traktować jako strumień (ewentualnie wiele strumieni), pakiety dowolnej wielkości, czy też małe pakiety (mieszczące się w jednym pakiecie UDP lub IP).

4.1.1. Małe pakiety

Przyjęcie małych pakietów jako nośnika danych utrudnia wykorzystanie protokołu w praktyce. Programista używający takiego protokołu sam musiałby zadbać o łączenie pakietów w większe. Efekt byłby taki, że w prawie każdym zastosowaniu konieczna byłaby dodatkowa warstwa niwelująca braki przyjętego rozwiązania, a sam protokół nie mógłby bazować na fakcie wysyłania większych pakietów lub strumieni danych.

4.1.2. Pakiety dowolnej wielkości

Często w pracach teoretycznych przyjmuje się założenie, że jest przesyłana ściśle określona, niezbyt duża (mieszcząca się w pamięci operacyjnej) ilość danych. Dzięki temu otrzymanie przez klienta danych jest tożsame z ich wykorzystaniem, w tym sensie, że serwer może przystąpić do wysyłania kolejnych, ponieważ klient może na bieżąco otrzymane dane umieszczać we wcześniej zaalokowanym buforze.

Rozwiązanie to ma jednak bardzo poważne wady:

- Serwer musi z góry znać rozmiar danych.
- Przed rozpoczęciem wysyłania danych ich rozmiar musi zostać przesłany do klientów, którzy z kolei muszą dostarczyć do serwera potwierdzenia. Ogranicza to mocno wydajność przy wysyłaniu wielu krótkich pakietów.
- Próba ominięcia powyższych ograniczeń wymaga stworzenia dodatkowej warstwy ukrywającej fakt przesyłania danych w pakietach określonego rozmiaru. W takim razie dlaczego tego nie zrobić od razu?

4.1.3. Strumienie

Strumienie wydają się nie mieć powyższych wad. Przesyłanie dowolnych pakietów można zrealizować wysyłając najpierw ich rozmiar.

Przy okazji częściowo zostaje rozwiązany problem API – można wykorzystać istniejące klasy *InputStream* oraz *OutputStream*, co ułatwia zastosowanie protokołu w praktyce.

Powyższe obserwacje skłoniły mnie do wyboru tego rozwiązania.

4.2. Połączenia i kanały

Kolejnym problemem okazało się równoległe przesyłanie kilku strumieni danych. Możliwe są trzy rozwiązania.

4.2.1. Jeden strumień na połączenie

Najprostszym rozwiązaniem jest nie rozważanie tego problemu przy projektowaniu protokołu. Równolegle można przesyłać dane oddzielnymi połączeniami.
Plusy i minusy:

+ znaczne uproszczenie protokołu,

— problem nie jest rozwiązywany, lecz przenoszony na inny poziom,

— zwiększenie liczby potwierdzeń przesyłanych przez klientów (dla każdego połączenia potwierdzenia przesyłane są osobno),

— większe zużycie zasobów – każde połączenie wymaga osobnych gniazd, buforów, wątków, itp.

4.2.2. Wiele równoległych strumieni – kanały

Kolejnym możliwym rozwiązaniem jest wyodrębnienie kanałów w każdym połączeniu. Każdym kanałem można wysyłać w danym momencie tylko jeden strumień. Połączeniem można przesyłać na przemian dane z kilku kanałów, niezależnie od długości strumieni.
Plusy i minusy:

+ znaczna komplikacja protokołu,

+ problem jest rozwiązywany, a nie przenoszony na inny poziom,

+ zmniejszenie liczby potwierdzeń przesyłanych przez klientów – w jednym pakiecie klient może przesłać potwierdzenia dotyczące wszystkich kanałów połączenia,

+ potencjalnie mniejsze zużycie zasobów – wystarczy jeden komplet gniazd, buforów systemu operacyjnego i wątków do obsługi wszystkich kanałów połączenia,

— może dochodzić do konfliktów w wykorzystaniu wspólnych zasobów – przede wszystkim bufora systemu operacyjnego.

4.2.3. Strumienie wysyłane oddzielnymi połączeniami, potwierdzenia wspólnym

Trzecie rozwiązanie polega na połączeniu dwóch poprzednich. Sprowadza się do wysyłania strumieni oddzielnymi połączeniami, natomiast potwierdzeń jednym wspólnym.

Powstaje niestety problem wysyłania żądań potwierdzeń. Trzeba się zdecydować na jedno z rozwiązań:

- Żądania wysyłane oddzielnym połączeniem – powoduje zwiększenie zasobów oraz tracimy zależność między dostarczeniem danych a dostarczeniem żądania (żądanie może dotrzeć wcześniej niż dane).
- Żadania wysyłane dowolnym z połączeń razem z danymi. Działanie jednego połączenia może negatywnie wpłynąć na pozostałe. Nadal brak zależności między dostarczenie danych i żądania.
- Przesyłanie żadań odpowiednimi połączeniami. Klient musiałby opóźniać przekazywanie odpowiedzi, aby zmniejszyć ich liczbę. Rozwiążanie to upodabnia się do zupełnego rozdzielenia strumieni.

Ostatecznie zdecydowałem się na rozwiązanie z punktu 4.2.2.

4.3. Numeracje i potwierdzenia

Najpoważniejszym problemem było zdecydowanie co i jak należy numerować oraz co i jak należy potwierdzać przy przyjętych założeniach, czyli możliwości przesyłania danych wieloma kanałami równolegle oraz kolejnych strumieni dowolnej wielkości skierowanych do dowolnej grupy klientów.

4.3.1. Dane

Z całą pewnością konieczne jest oznaczanie samych danych. Protokół TCP/IP ([RFC793]) numeruje kolejne bajty przesyłanych danych. Umożliwia to ich przesyłanie i retransmisje w pakietach różnej wielkości. Alternatywnym rozwiązaniem jest numerowanie wysyłanych fragmentów, czyli zgrupowanych danych. Uniemożliwia to zmianę rozmiaru fragmentu przy retransmisji, co jednak nie jest istotnym ograniczeniem ze względu na małe prawdopodobieństwo wystąpienia retransmisji. W niniejszej pracy przyjąłem to drugie rozwiązanie, ponieważ umożliwia ono przeznaczenie mniejszej liczby bitów na zakodowanie numeru.

Kolejnym pytaniem, na które musiałem odpowiedzieć to czy numerację danych prowadzić niezależnie od strumieni, czy też rozpoczęcie nowego strumienia ma rozpoczynać numerację danych od nowa. Odpowiedź na to pytanie stała się oczywista po przemyśleniu w jaki sposób klient ma potwierdzać dane. Przyjęcie, że w odpowiedzi klienta jest numer fragmentu, którego jeszcze nie dostał (analogicznie do TCP/IP – numer sekwencji, której jeszcze nie otrzymał), sprawia, że konieczna jest znajomość numeru pierwszego fragmentu kolejnego strumienia. Można ją zapewnić jedynie przez przyjęcie, że rozpoczęcie kolejnego strumienia rozpoczyna numerację danych od początku.

4.3.2. Strumienie

Numeracja strumieni jest konieczna, chociażby po to, aby odróżnić dane pochodzące z różnych strumieni. Gdyby jednym z wymagań nie było zachowanie kolejności przy dostarczaniu

danych, to jedynym wymaganiem stawianym numeracji strumienia byłoby zapewnienie, że dwa strumienie wysyłane w ciągu określonego przedziału czasu nie mogą mieć takich samych numerów. Wspomniane wymaganie sprawia, że powinna być możliwość uszeregowania strumienia w kolejności ich wysyłania na podstawie tych numerów. Krótko mówiąc strumieniom należy nadawać numery kolejne.

Numer kolejny strumienia razem i numer kolejny fragmentu służą do identyfikacji danych, a więc muszą być wysyłane w każdym fragmencie. Dzięki nim klient będzie w stanie stwierdzić czy zgubił jakieś dane.

4.3.3. Indywidualna numeracja strumieni każdego klienta

Kolejny problem wynika z założeń, że różni klienci mogą zamówić różne strumienie oraz z wymagania dostarczenia wszystkich danych w kolejności ich wysłania. Rozważmy następujący przykład.

1. Serwer wysyła bardzo krótkie strumienie o numerach 1, 2, 3, 4 oraz 5 i chce mieć gwarancję, że klient otrzyma strumienie 1, 3 oraz 5.
2. Przy przesyłaniu gubiony jest w całości strumień 3.
3. Klient otrzymuje wszyskie pozostałe strumienie, z których przetwarza jedynie 1 i 5, bo tylko one są dla niego interesujące.

Nasuwają się dwa pytania.

1. Co zrobić, aby klient wiedział, że zgubił strumień 3?
2. Co zrobić, aby klient nie zaczął przetwarzać strumienia 5 przed rozpoczęciem przetwarzania strumienia 3?

Jako pierwsze nasuwa się takie oto rozwiązanie.

Co jakiś czas serwer wysyła prośbę o przekazanie stanu przez klientów. Wydaje mi się, że nie ma sensu wysyłać takiej prośby prywatnym kanałem, gdyż narzut byłby zbyt duży (dodatkowy pakiet lub dwa na każdego klienta). Wystarczy dołączyć kilka bajtów do pakietu wysyłanego otwartym kanałem przy okazji wysyłania danych.

Klient musi pamiętać numery zamówionych strumieni, które otrzymał, oraz ile pakietów w obrębie tych strumieni dotarło. Ponadto pamięta największy numer strumienia oraz fragmentu w obrębie tego strumienia, który otrzymał.

Odpowiedzią na pytanie o stan jest wysłanie prywatnym kanałem powyższych danych dla każdego kanału. Na ich podstawie serwer będzie w stanie stwierdzić czy jakiś zamówiony przez klienta strumień nie został pominięty w całości.

Klient może usunąć zapamiętaną listę po upewnieniu się, że serwer odpowiedź otrzymał – jeśli kanał prywatny będzie zrealizowany przez połączenie TCP/IP, to wystarczającą pewność daje nam wysłanie wspomnianej listy.

Wadą powyższego rozwiązania jest brak pewności dostarczenia do klienta strumieni w takiej kolejności w jakiej zostały wysłane, czyli brak odpowiedzi na drugie postawione pytanie. Sposobów uniknięcia tej wady jest kilka:

- Wstrzymanie wysyłania przez serwer kolejnego strumienia do czasu potwierdzenia otrzymania nagłówka poprzedniego zamówionego strumienia przez każdego klienta. Minusem jest znaczne ograniczenie wydajności protokołu – w skrajnym przypadku okno (maksymalna liczba pakietów wysłanych a nie potwierdzonych) będzie miało wielkość 1, co jest nieakceptowalne.

- Uniemożliwienie tworzenia tak małych strumieni. Nie wiadomo co tak naprawdę oznacza „małych" – nie da się tego oszacować, gdyż między pakietem 1. a 3. oraz między 3. a 5. może być ich dowolnie dużo.

- Wprowadzenie dodatkowej numeracji strumieni indywidualnej dla każdego klienta – wówczas wystarczy, że klient będzie pamiętał największy zamówiony strumień, który zaczął przyjmować oraz największy numer strumienia i największy numer fragmentu strumienia, który otrzymał. Szczegóły zostaną przedstawione w dalszej części pracy.

4.3.4. Kiedy wysyłać numery strumieni klientów

Nie można wysyłać wszystkich numerów strumienia w każdym fragmencie, ponieważ numery te stanowiłyby zbyt dużą ich część. Nie ma też takiej potrzeby, ponieważ klienci mogą rozpocząć przetwarzanie strumienia tylko od początku (albo punktu synchronizacji, ale o tym w dalszej części pracy). Wystarczy zatem wysyłać wszystkie numery jedynie w pierwszym fragmencie.

4.4. Sposób wysyłania danych

Istnieją dwie możliwości wysyłania danych: rozgłaszanie oraz rozsyłanie grupowe. Ponadto trzeba zdecydować jakich bibliotek używać. W Javie mamy do wyboru gniazda oraz kanały (ang. *channels*).

Kanały stanowią niewątpliwą pokusę, ponieważ są potencjalnie szybsze i istnieje możliwość zminimalizowania liczby kopiowań dzięki zastosowaniu buforów. Ponadto pozwalają na pracę asynchroniczną, co zmniejsza zapotrzebowanie na wątki i ułatwia programowanie.

Gniazda nie mają żadnej z powyższych zalet, jednak to na nie się zdecydowałem. Oto przyczyny:

- Brak w kanałach wsparcia dla rozsyłania grupowego. Przewiduje się, że zostanie ono wprowadzone dopiero w wersji 1.6 Javy ([JSR203]).

- Błędy i ograniczenia występujące zwłaszcza w trybie pracy asychronicznej. Oficjalnie implementacja kanałów w wersji 1.4.2 Javy nie jest kompletna. Między innymi biblioteki nie ukrywają w wystarczającym stopniu różnic miedzy systemami operacyjnymi. Przykładem może być sposób wznawiania wątków oczekujących na zdarzenia ([TNC]).

- Użycie kanałów zmniejszyłoby prawdopodobnie o jedno liczbę kopiowań pamięci. Moja pierwsza implementacja protokołu nie kładzie na to zbyt mocnego nacisku.

Jeśli chodzi o wybór między rozgłaszaniem a rozsyłaniem grupowym, to w pierwszej kolejności zdecydowałem się na drugą opcję. W przyszłości będzie można zaimplementować rozwiązanie z wykorzystaniem rozgłaszania i je porównać.

Rozdział 5

Protokół

Protokół opisany w tym rozdziale zaprojektowałem kierując się wymaganiami sprecyzowanymi w rozdziale 3. Wydaje mi się, że większość z nich udało mi się spełnić w stopniu co najmniej zadowalającym. Jednym z podstawowych wymagań była odpowiednia wydajność – większa niż w przypadku rezygnacji z rozgłaszania. O tym czy zostało ono spełnione, na podstawie pierwszej, dalekiej od doskonałości implementacji, można dowiedzieć się z rozdziału 8.

Informacje zawarte w niniejszym rozdziale pozwalają na wykonanie własnej implementacji tego protokołu niezawodnego rozgłaszania (PNR).

5.1. Związek z innymi protokołami

Istnieje już wiele protokołów niezawodnego rozgłaszania. Zdecydowałem się jednak na zaprojektowanie własnego dlatego, że żaden z nich nie spełniał wymagań postawionych w rozdziale 3.

Protokoły FEC dobrze radzą sobie w przypadku gubienia pojedynczych pakietów, natomiast zupełnie nie zdają egzaminu w przypadku gubienia setek kolejnych pakietów z powodu wstrzymania działania procesu obsługującego protokół.

Protokoły NAK mają zbyt duże wymagania pamięciowe lub ograniczają niezawodność ([PGM]).

Protokoły podstawowe ACK nie dają wystarczających korzyści – ich skalowalność jest zbyt mała. Jedyną przewagą protokołów drzewiastych nad postawowymi w sieciach lokalnych jest mniejsze obciążenie serwera (nie musi obsługiwać odpowiedzi od wszystkich klientów). Okupione jest to jednak dużym zwiększeniem stopnia komplikacji implementacji – serwer musi zarządzać strukturą drzewa.

Część proponowanych rozwiązań przyjmuje uproszczony model, w którym nie rozróżnia potwierdzenia przez klienta otrzymania danych od ich przetworzenia. Jeśli w takim modelu potwierdzenie przez klienta oznacza potwierdzenie przetworzenia danych, to może dochodzić do zbędnych retransmisji. Jeśli przyjąć natomiast, że potwierdzenie oznacza potwierdzenie przyjęcia danych (zapisania ich buforze), to powstaje niebezpieczeństwo gubienia danych przez klienta z powodu braku miejsca w buforze odbiorczym.

Inne propozycje ([Lane]) zakładają, że rozmiar przesyłanych danych jest z góry znany. Rozwiązanie takie nie jest wystarczająco uniwersalne, gdyż utrudnia przesyłanie strumieni danych. Rozgłaszanie ich nadal jest możliwe, ale wymaga sztucznego opóźniania wysyłania danych i dzielenia ich na bloki.

Żadne ze znanych mi rozwiązań nie oferuje możliwości równoległego przesyłania kilku strumieni. Jest to oczywiście możliwe, ale wymaga utworzenia dodatkowych połączeń. Żadne też

nie kładzie w takim stopniu nacisku na zachowanie maksymalnej wydajności oraz kolejności dostarczenia danych przy przesyłaniu strumieni zarówno bardzo krótkich, jak i o nieograniczonym rozmiarze.

Ostatecznie doszedłem do wniosku, że przy postawionych wymaganiach najlepiej może się sprawdzać protokół będący wypadkową ACK grupującego potwierdzenia i NAK z ankietowaniem. PNR nie jest protokołem NAK, ponieważ nie zakłada, że klienci będą od razu zgłaszać fakt zgubienia danych, gdyż mogą nie mieć wystarczających do tego informacji. Bazuje jednak na ankietowaniu. Zakłada dwa sposoby udzielania odpowiedzi na ankiety: natychmiastowe (tak jak w protokołach NAK), oraz opóźnione, będące odpowiednikami potwierdzeń występujących w protokołach ACK.

5.2. Topologia i role

Protokół jest przeznaczony do pracy w sieci lokalnej, czyli takiej, w której węzły są ze sobą bezpośrednio połączone szybkim łączem lub łączami, oraz istnieje możliwość wysłania tej samej informacji do wszystkich węzłów jednocześnie. Dokładnie jeden węzeł – serwer – jest wyróżniony. Jest on źródłem danych rozsyłanych do pozostałych węzłów – klientów.

Serwer jest stroną aktywną – on decyduje co i kiedy wysłać. On też decyduje, które dane i do których klientów są skierowane. Klienci wysyłają odpowiedzi na ankiety, które serwer interpretuje jako potwierdzenia otrzymania lub przetworzenia danych albo informacje o zagubieniu danych.

5.3. Czego nie obejmuje protokół

Niniejszy protokół nie obejmuje kwesti takich jak:

- Zarządzanie sesjami klientów. Protokół zakłada, że powiązanie adresu klienta z jego identyfikatorem, wykorzystywanym w protokole, w jakiś sposób się odbędzie i informacja ta dotrze do serwera oraz klienta. Z punktu widzenia protokołu nie jest istotne jak.
- Bezpieczeństwo. Protokół ten był projektowany z myślą o zastosowaniu w sieci lokalnej, w związku z czym nie zawiera żadnych mechanizmów zabezpieczających przed atakami.

5.4. Protokoły warstw niższych

Protokół bazuje na dwóch protokołach warstw niższych. Każdy z nich jest wykorzystywany do komunikacji w jedną stronę. Serwer rozgłasza dane do wszystkich klientów równocześnie, natomiast klienci pojedynczo przesyłają dane do serwera.

5.4.1. Protokoły służące do przesyłania danych do serwera

Protokół warstwy niższej służący do przesyłania danych od klienta do serwera musi spełniać następujące warunki:

- umożliwiać przesyłanie danych w pakietach. Podstawową jednostką PNR jest pakiet.
- dostarczać pakiety w takiej kolejności, w jakiej zostały wysłane. Dopuszczalna jest sporadyczna zmiana kolejności dostarczenia – protokół traktuje pakiety dostarczone za późno jako zagubione.

- umożliwiać przesłanie w całości pakietów o wielkości od kilkudziesięciu bajtów do minimum kilobajta. Jeśli stosowane jest dzielenie pakietów na mniejsze części przy przesyłaniu, to proces ten musi być przezroczysty, tzn. nieodróżnialny od podanego wymagania.

- zapewniać kontrolę błędów. PNR nie wprowadza własnych mechanizmów służących do wykrywania przekłamań podczas transmisji.

Powyższe warunki spełniają zarówno protokoły warstwy łącza (Ethernet, IEEE 802.3 MAC), jak i warstw wyższych, takie jak IP, IPX czy UDP. Jestem przekonany, że protokół można zaimplementować na bazie dowolnego z nich, chociaż w dalszej części pracy mogę się posługiwać pojęciami wskazującymi na zastosowanie protokołu UDP.

5.4.2. Protokoły służące do rozsyłania danych przez serwer

Protokół warstwy niższej służący do rozsyłania danych przez serwer musi spełniać wszystkie warunki podane powyżej (jak do trasmisji w drugą stronę), a ponadto:

- Musi umożliwiać przesłanie pakietu równocześnie do wszystkich klientów. Gdyby to wymaganie nie było spełnione, to zastosowanie PNR nie miałoby sensu, gdyż protokół, będący tematem tej pracy, nie mógłby być lepszy od wysyłania danych do każdego klienta osobno.

Warunek ten spełniony jest przez wszystkie wspomniane wcześniej protokoły, a zatem może to być ten sam protokół. W przypadku protokołu UDP mamy dodatkowo do wyboru rozgłaszanie (do wszystkich klientów) oraz rozsyłanie grupowe (tylko do zainteresowanych). W celu łatwiejszego zrozumienia działania PNR założę, że protokół będzie implementowany na bazie rozsyłania grupowego.

5.5. Numery i operacje na nich

Można założyć, że numery oraz identyfikatory są nieujemnymi liczbami całkowitymi. Identyfikatory są jedynie porównywane w celu stwierdzenia czy są sobie równe, w związku z czym zbiór ich wartości jest w naturalny sposób ograniczony. Interesującym wynikiem porównywania dwóch numerów jest natomiast stwierdzenie czy są równe, a jeśli nie to który z nich jest większy. Zakładam, że zbiór nie jest ograniczony, co od razu nasuwa podejrzenie, że nie jest możliwa implementacja. Na szczęście tak nie jest. Można założyć, że numery są liczbami całkowitymi n-bitowymi bez znaku. Wszystkie operacje przeprowadzane są na nich modulo 2^n. W większości przypadków operację porównania numerów N i M można zdefiniować następująco:

- jeśli $M - N = 0$ to numery są równe,

- jeśli $M - N < 2^{n-1}$ to numer N jest mniejszy od M,

- jeśli $M - N >= 2^{n-1}$ to numer N jest większy od N.

Do porównania niektórych numerów potrzebny jest szerszy kontekst. Na przykład do porównywania numerów strumieni można wykorzystać numery pakietów. Wówczas operacja porównywania może dać wynik „nie wiadomo”.

5.6. Pojęcia

Do zrozumienia budowy i zasad działania protokołu konieczne jest zapoznanie się z używanymi w pracy pojęciami, takimi jak połączenie, kanał, strumień czy fragment. W kolejnych punktach znajdują się ich krótkie opisy.

5.6.1. Połączenie

Połączenie reprezentuje możliwość rozesłania danych przez serwer do klientów oraz otrzymanie od nich odpowiedzi. Czym innym jest połączenie dla serwera i klienta. Dla łatwiejszego zrozumienia zakładam, że protokołem wartwy niższej jest UDP.

Dla serwera połączenie to para złożona z:

- gniazda skonfigurowanego do rozsyłania danych na dany adres rozsyłania grupowego i port,
- gniazda skonfigurowanego do odbierania pakietów na danym porcie nadawanych przez klientów.

Dla klienta połączenie to para złożona z:

- gniazda skonfigurowanego do odbierania danych nadawanych przez serwer na dany adres rozsyłania grupowego i port,
- gniazda skonfigurowanego do wysyłania pakietów na adres serwera i dany port.

Połączenia wszystkich klientów powinny być skonfigurowane identycznie.

Z każdym gniazdem związany jest bufor nadawczy i odbiorczy. Rozmiar buforów nadawczych nie jest istotny – ważne jest tylko, aby nie wprowadzał on zbyt dużych opóźnień. Rozmiary buforów odbiorczych (serwera i klientów) są bardzo istotne i określone w dalszej części tego rozdziału. Zmiana ich rozmiarów może znacząco wpłynąć na wydajność.

5.6.2. Pakiet

Pakiet jest grupą danych przesyłanych połączeniem niepodzielnie i jednorazowo. Maksymalny rozmiar pakietu jest określony przez protokół warstwy niższej. W przypadku IEEE 802.3 wynosi on około 1500 bajtów, natomiast dla UDP są to 64 kilobajty.

Każdy pakiet wysyłany przez serwer ma swój numer kolejny, na który powołuje się klient przy przekazywaniu potwierdzeń. Służy on również do wykrywania zagubionych danych, zarówno przez serwer, jak i przez klienta.

Serwer nigdy nie wysyła dwóch takich samych pakietów. Pakiety z retransmitowanymi danymi są nieodróżnialne (pod względem struktury) od pakietów z danymi wysyłanymi po raz pierwszy. Jedynie klienci, którzy już wcześniej otrzymali pakiet z tymi samymi danymi są w stanie stwiedzić, że zawiera on retransmitowane dane. Co więcej, w jednym pakiecie mogą być przesyłane zarówno nowe, jak i stare dane.

Sens numeru pakietu wysyłanego przez klienta jest inny. Jest to numer pakietu, jaki klient ostatnio otrzymał od serwera. W związku z tym możliwe jest wysłanie przez klienta dwóch pakietów o tym samym numerze, ale innej zawartości.

5.6.3. Kanał

Wydzielona część połączenia to kanał. Dzięki wydzieleniu kanałów w obrębie połączenia możliwe jest równoległe przesyłanie niezależnych danych, skierowanych do różnych klientów. Wydzielenie polega na przesyłaniu na przemian danych z różnych kanałów tym samym połączeniem. W przypadku konieczności przesłania danych z kilku kanałów równocześnie, o pierwszeństwie decyduje wiek danych – starsze są wysyłane wcześniej.

Można oczywiście ograniczyć się do jednego kanału na połączenie i równoległą transmisję niezależnych danych przeprowadzać oddzielnymi połączeniami. Powoduje to jednak zwiększone zużycie zasobów (gniazd i co za tym idzie buforów gniazd). Przesyłanie danych z różnych kanałów jednym połączeniem ma jeszcze jedną zaletę – klienci mogą grupować odpowiedzi na ankiety z różnych kanałów i wysyłać je w jednym pakiecie.

Z każdym kanałem związany jest bufor po stronie serwera i bufor po stronie klienta. Na serwerze dane przed wysłaniem są umieszczane w buforze kanału, natomiast usuwane z niego dopiero po potwierdzeniu przekazania ich aplikacjom klienckim przez implementacje protokołu po stronie wszystkich zainteresowanych nimi klientów. Jeśli istnieje taka potrzeba, to dane te mogą być wielokrotnie retransmitowane.

Po otrzymaniu danych klient umieszcza je w buforze kanału. Usunięcie odbywa się dopiero po przekazaniu ich aplikacji, albo wcześniej, jeśli dane nie są zamówione i istnieje potrzeba zwolnienia miejsca w buforze na nowe dane.

Każdy kanał ma swój unikatowy identyfikator w obrębie połączenia.

5.6.4. Strumień

Strumieniem jest uporządkowany zbiór danych przesyłanych w obrębie kanału. W danym momencie może być przesyłany tylko jeden strumień danych w obrębie jednego kanału (wyjątkowo, w przypadku retransmisji, może dochodzić do przesyłania danych z kilku strumieni tego samego kanału).

Każdy strumień ma swój początek i może, ale nie musi, mieć swój koniec. Ponadto w każdym strumieniu może wystąpić dowolna liczba tzw. punktów synchronizacji, czyli miejsc, od których klienci mogą zacząć odbierać dane, jeśli nie zdążyli od początku. Początek strumienia można traktować jako pierwszy punkt synchronizacji strumienia.

Serwer może zapisywać klientów na strumienie. Od zapisanego klienta wymagane są potwierdzenia otrzymanych danych. Serwer dba o to, aby każdy zapisany klient otrzymał wszystkie dane począwszy od pierwszego punktu synchronizacji po zapisaniu klienta aż do końca strumienia. Nie przewidziałem możliwości wypisania klienta, ale jeśli będzie taka potrzeba, to można łatwo zmodyfikować protokół tak, aby było to dopuszczalne.

Każdy strumień ma przynajmniej jeden numer – numer kolejny w obrębie kanału. Dodatkowo ma tyle numerów kolejnych, ile klientów jest na niego zapisanych (czyli dla danego strumienia liczba ta może się zwiększać w czasie w punktach synchronizacji).

W celu ułatwienia podsłuchiwania oraz zabezpieczenia się przed przetwarzaniem przez klientów strumieni w innej kolejności niż zostały wysłane, w każdym punkcie synchronizacji przesyłane są numery kolejne strumieni dla klientów, którzy na niego nie zapisali się, ale zapisali się na wcześniejsze. Zmiana wartości numeru kolejnego strumienia dla klientów następuje po zamknięciu zamówionego strumienia.

Przykład

Załóżmy, że strumienie nie mają punktów synchronizacji (poza początkiem). Na pierwszy strumień zapisali się klienci K1 i K2. Na drugi nie zapisał się nikt, a na trzeci klienci K2 i K3.

Wówczas strumienie będą miały następujące numery:

- Strumień pierwszy – numer kolejny to 0, zarejestrowani klienci to K1 i K2, numer dla klienta K1 – 0, numer dla klienta K2 – 0;
- Strumień drugi – numer kolejny to 1, zarejestrowanych klientów brak, numer dla klienta K1 – 1, numer dla klienta K2 – 1,
- Strumień trzeci – numer kolejny to 2, zarejestrowani klienci to K2 i K3, numer dla klienta K1 – 2 numer dla klienta K2 – 1, numer dla klienta K3 – 0.

5.6.5. Fragment

Fragment jest częścią strumienia przesyłaną w całości w jednym pakiecie. W skład pakietu mogą wchodzić fragmenty z kolejnych strumieni oraz z różnych kanałów, oczywiście pod warunkiem, że są z jednego połączenia.

Punkt synchronizacji wymusza powstanie nowego fragmentu. Fragmenty tworzone są przez serwer zaraz przed wysyłaniem – ze strumienia wydzielana jest część, którą można zmieścić w jednym pakiecie. Fragmenty po wysłaniu nie są już dalej dzielone ani łączone w większe.

Fragment ma swój numer kolejny w obrębie strumienia. Jest on używany w protokole do adresowania danych strumienia. Na niego powołuje się klient przy potwierdzaniu danych oraz weryfikacji kompletności oraz prawidłowej kolejności otrzymywania danych.

Zgodzę się z tezą, że jest to pewne ograniczenie, które może negatywnie wpływać na wydajność. Lepszym rozwiązaniem mogłoby być posługiwanie się adresem danych strumienia (ang. *offset*) na wzór protokołu TCP i każdorazowe tworzenie fragmentów jedynie w celu ich umieszczenia w pakiecie, jednak obawiam się, że wpłynęłoby to negatywnie na poziom komplikacji implementacji.

5.7. Budowa pakietu

Asymetryczność protokołu sprawia, że pakiety wysyłane przez serwer i przez klientów całkowicie się różnią. Nie zdefiniowałem dokładnie rozmiaru poszczególnych pól, gdyż uznałem, że nie jest to aż tak istotne. Praca ta nie ma na celu ustalenia binarnego standardu, a jedynie przedstawienie pewnego pomysłu i udowodnienie, że jego realizacja jest możliwa.

Definicja pakietów sprowadza się do podania ich gramatyki, zdefiniowania pól oraz zgrubnego oszacowania ich wielkości. Nie należy zwracać uwagi na niejednoznaczność podanych gramatyk.

5.7.1. Pakiet rozsyłany przez serwer

Zakładam, że protokół niższej warstwy zapewnia jednoznaczną identyfikację serwera. Jeśli tak nie jest, to odpowiednią informację należy dodatkowo umieścić w pakiecie lub konfiguracji klienta.

$$Packet ::= P_{SEQ}, (Fragment)*, (Status_{REQ})*$$

$$Fragment ::= Channel_{Id}, S_{SEQ}, F_{SEQ}, F_{OPTS}, Data$$

$$F_{OPTS} ::= (SYNC, Clients_{OLD}, Clients_{NEW}, Clients_{KNOWN})?, EOS?$$

$$Clients_{OLD} ::= (Client_{Id}, CS_{SEQ})*$$

$$Clients_{NEW} ::= (Client_{Id}, CS_{SEQ})*$$
$$Clients_{KNOWN} ::= (Client_{Id}, CS_{SEQ})*$$
$$Status_{REQ} ::= Client_{Id}$$

Znaczenie poszczególnych symboli:

- $Packet$ — cały pakiet,
- P_{SEQ} — numer kolejny pakietu,
- $Fragment$ — fragment,
- $Status_{REQ}$ — żądanie przesłania przez klienta swojego stanu (ankieta),
- $Channel_{Id}$ — identyfikator kanału,
- S_{SEQ} — numer kolejny strumienia,
- F_{SEQ} — numer kolejny fragmentu,
- F_{OPTS} — dodatkowe znaczniki fragmentu,
- $SYNC$ — znacznik punktu synchronizacji oraz początku pakietu, umieszczany w pierwszym fragmencie strumienia oraz pierwszym fragmencie po punkcie synchronizacji,
- EOS — znacznik końca strumienia, umieszczany w ostatnim fragmencie strumienia,
- $Clients_{OLD}$ — lista klientów, którzy już wcześniej byli zapisani na ten strumień,
- $Clients_{NEW}$ — lista nowych klientów tego strumienia,
- $Clients_{KNOWN}$ — lista pozostałych klientów tego kanału (klienci, którzy zamówili wcześniejsze strumienie ale nie zamówili tego),
- CS_{SEQ} — numer kolejny strumienia wg numeracji klienta,
- $Data$ — blok danych strumienia,
- $Client_{Id}$ — identyfikator klienta.

5.7.2. Pakiet z odpowiedzią klienta

Pakiet z odpowiedzią klienta ma dużo prostszą strukturę.

$$Packet ::= P_{SEQ}, Client_{Id}, (Channel_{STATUS})*$$
$$Channel_{STATUS} ::= Channel_{Id}, Fragment_{RCVD}, Fragment_{RLSD}$$
$$Fragment_{RCVD} ::= CS_{SEQ}, F_{SEQ}$$
$$Fragment_{RLSD} ::= CS_{SEQ}, F_{SEQ}$$

Znaczenie poszczególnych symboli:

- $Packet$ — cały pakiet,
- P_{SEQ} — numer pakietu ostatnio otrzymanego przez klienta,

- $Client_{Id}$ — identyfikator klienta,
- $Channel_{STATUS}$ — stan bufora kanału,
- $Channel_{Id}$ — identyfikator kanału,
- $Fragment_{RCVD}$ — dane fragmentu zamówionego strumienia, który jako następny zostanie dodany do bufora (klient spodziewa się, że jako następny otrzyma ten fragment),
- $Fragment_{RLSD}$ — dane fragmentu zamówionego strumienia, który jako następny zostanie usunięty z bufora (po przekazaniu go aplikacji klienta),
- CS_{SEQ} — numer kolejny strumienia wg numeracji klienta,
- F_{SEQ} — numer kolejny fragmentu.

5.8. Serwer

Protokół opiera się na kilku podstawowych zasadach:

- Całym protokołem zarządza serwer – on decyduje co, do kogo i kiedy wysłać. Ułatwia to zapanowanie nad całością. Wyjątkiem jest potwierdzanie przez klientów przetworzenia danych.
- Kanały są obsługiwane niezależnie – nie śledzimy i nie przewidujemy obciążenia zasobów wspólnych takich jak bufory połączeń klientów.
- Serwer dba o to, aby żaden klient nie gubił danych z powodu przepełnienia bufora kanału – o jego wielkości decyduje stała $Channel_{BUFSIZE}$, będąca jednocześnie rozmiarem okna, czyli ilością danych, które w danym kanale zostały wysłane, a nie zostały przez wszystkich potwierdzone.
- Klienci, którzy nie zapisali się na dany strumień nie wpływają na zachowanie serwera. Mogą otrzymywać dane z tego strumienia (podsłuchiwać), ale w przypadku zgubienia danych nie mogą liczyć na pomoc serwera w ich odzyskaniu. Jedynym wyjątkiem jest przesyłanie numerów kolejnych strumieni w punktach synchronizacji dla klientów, którzy w przeszłości zapisali się na co najmniej jeden strumień.

5.8.1. Zarys

1. Aplikacja na serwerze poprzez dostarczone API umieszcza dane w buforze cyklicznym. Jeśli nie ma wystarczającej ilości miejsca, to aplikacja jest zawieszana do momentu jego zwolnienia. Umieszczenie danych w buforze powoduje obudzenie wątku obsługującego protokół.
2. Wątek sprawdza czy są odpowiedzi klientów (ze statusami). Jeśli tak, to na ich podstawie uaktualnia swoje struktury (np. zwalniając część danych z bufora i tym samym budząc aplikację korzystającą z API).
3. Wątek przygotowuje pakiet do wysłania. W tym celu wybiera odpowiednie fragmenty (do retransmisji) lub tworzy nowe. Kieruje się wiekiem danych oraz statusem klientów.
4. Wątek uzupełnia pakiet o żądania przesłania statusu przez klientów (ankiety) oraz – jeśli jest coś do wysłania – wysyła pakiet.

5. Jeśli nie ma nic do wysłania, to zasypia do czasu zajścia jakiegoś zdarzenia. Jeśli do wysłania są jedynie zapytania o status, to zasypia na chwilę. Jeśli do wysłania są dane, to nie zasypia, tylko rozpoczyna pracę od nowa.

6. Wątek może zostać obudzony poprzez wstawienie danych do bufora cyklicznego oraz odebranie odpowiedzi od klienta.

5.8.2. Status klienta

Serwer przechowuje następujące dane o każdym kliencie:

- Identyfikator klienta. Protokół nie obejmuje metody jego nadawania.
- Numer pakietu ostatnio otrzymanego przez klienta (P_{SEQ}).
- Dla każdego kanału, którym klient jest lub był zainteresowany (jest lub był zapisany na strumień kanału):
  - Numer tego kanału ($Channel_{Id}$).
  - Ostatnio nadany numer strumienia klienta (CS_{SEQ}).
  - Dla każdego fragmentu strumienia zamówionego przez klienta, znajdującego się w buforze kanału – numer kolejny strumienia klienta oraz informację czy fragment ten został już przez klienta potwierdzony (rozróżniamy potwierdzenie otrzymania i przetworzenia).

5.8.3. Bufor kanału

Bufor kanału jest buforem cyklicznym. Podzielony jest na dwie części.

Pierwsza część zawiera fragmenty, które zostały już wysłane do klienta, ale nie zostały jeszcze przez niego potwierdzone (tzn. dane z tego fragmentu nie zostały jeszcze przekazane aplikacji z nich korzystającej).

Część ta ma zmienny rozmiar, który waha się od zera do stałej $Channel_{BUFSIZE}$ określającej jednocześnie rozmiar buforów kanałów wszystkich klientów. Jest to odpowiednik okna z protokołu TCP z tą jednak różnicą, że okno w PNR jest stałej wielkości. Wynika to z trudności, jaką byłoby uzgodnienie okna z wieloma klientami. Wartość tę należy dobrać empirycznie.

Druga część zawiera świeże dane, które nie zostały jeszcze przekształcone we fragmenty i wysłane. Aplikacja na serwerze poprzez API wstawia dane do tej części bufora. Wątek obsługujący protokół w miarę zwalniania miejsca w pierwszej części bufora przekształca dane z drugiej części na fragmenty i tym samym umieszcza je w części pierwszej.

5.8.4. Fragmenty

Z fragmentami znajdującymi się w pierwszej części wspomnianego wcześniej bufora związane są dodatkowe informacje, które nie są rozsyłane. Należą do nich:

- czas utworzenia fragmentu – jest wykorzystywany przy podejmowaniu decyzji o pierwszeństwie wysłania danych z różnych kanałów tym samym połączeniem,
- zbiór klientów, którzy zamówili strumień, ale jeszcze nie potwierdzili otrzymania tego fragmentu,

- zbiór, będący nadzbiorem powyższego zbioru, klientów, którzy zamówili strumień, ale jeszcze nie potwierdzili przekazania tego fragmentu aplikacji klienckiej (co jest równoznaczne z usunięciem tego fragmentu z bufora kanału klienta).

Powyższe zbiory aktualizowane są po otrzymaniu potwierdzeń od klientów.

5.8.5. Tworzenie i wysyłanie pakietu

Tworzenie i wysyłanie pakietu składa się z kilku etapów:

- wybrania odpowiednich fragmentów do wysłania,
- wybrania odpowiednich zapytań o status klientów (ankiet),
- zbudowania na podstawie powyższych danych pakietu,
- wysłania go.

Wybieranie fragmentów składa się z kilku etapów wykonywanych w pętli tak długo, aż wybrane fragmenty zapełnią większość pakietu, pozostawiając miejsce jedynie na zapytania o status, albo zabraknie fragmentów do wysłania.

1. Zaproponowanie co najwyżej jednego fragmentu z każdego kanału. Jeśli propozycja dotyczy nowego fragmentu, to jest on tworzony, ale nie do końca, gdyż nie ma jeszcze ustalonego rozmiaru. Przy okazji tworzone są propozycje zapytań o status. Szczegóły podano w punkcie 5.8.6.
2. Wybór najstarszego spośród zaproponowanych fragmentów.
3. Pobranie wybranego zaproponowanego fragmentu, wiążące się z ostatecznym ustaleniem jego rozmiaru (tylko w przypadku nowego fragmentu).

Wybieranie odpowiednich zapytań o status powinno się odbywać w oparciu o statystyki zebrane w dotychczasowym działaniu protokołu, czas ostatnich pytań o status oraz stopnia wypełnienia buforów klienta. W punkcie 5.8.7 zaproponowane zostało odpowiednie rozwiązanie.

5.8.6. Zaproponowanie fragmentu do wysłania

Zaproponowanie kolejnego fragmentu do wysłania dla danego kanału przebiega następująco:

1. Dla każdego fragmentu w kolejności od najstarszego do najnowszego powtarzaj punkty od 2 do 5.
2. Dla każdego klienta, który zamówił ten fragment ale jeszcze nie potwierdził jego otrzymania (numer ostatniego pakietu, jaki klient otrzymał jest mniejszy od numeru pakietu, w którym ostatnio ten fragment został wysłany) zgłoś potrzebę wysłania do niego ankiety z priorytetem proporcjonalnym do wieku fragmentu (szczegóły w punkcie 5.8.7).
3. Dla każdego klienta, który zamówił ten fragment i potwierdził jego otrzymanie ale jeszcze nie potwierdził jego przetworzenia zgłoś potrzebę wysłania do niego ankiety z priorytetem proporcjonalnym do wieku fragmentu, ale mniejszym niż w przypadku z punktu 2.

4. Jeśli którykolwiek klient, który zamówił ten fragment, potwierdził otrzymanie pakietu, w którym ostatnio został wysłany ten fragment, lub pakietu późniejszego, to zaproponuj ten fragment – trzeba go retransmitować. Jeśli fragment ten zostanie wybrany do wysłania, to uaktualnij w nim numer pakietu, w którym zostanie wysłany.

5. Jeśli fragment ten nie został jeszcze wysłany i jest na niego miejsce w części buforu z wysłanymi fragmentami (sumaryczny rozmiar wysłanych i nie potwierdzonych fragmentów plus minimalny rozmiar tego fragmentu jest nie większy niż rozmiar bufora klienta $Channel_{BUFSIZE}$), to go zaproponuj.

5.8.7. Wybór zapytań o status (ankiet) do wysłania

Przy wyznaczaniu klientów, których trzeba zapytać o status należy kierować się poniższymi spostrzeżeniami:

- należy minimalizować liczbę zapytań o status – nadmiarowe zapytania i odpowiedzi na nie powodują zmniejszenie wydajności protokołu,
- należy w miarę równomiernie rozkładać zapytania aby zapobiec sytuacji, w której wszyscy klienci jednocześnie będą próbowali udzielić odpowiedzi – w celu uniknięcia konfliktów,
- należy unikać pytań, na których odpowiedzi nic nie wniosą lub wniosą niewiele (np. dlatego, że spodziewamy się odpowiedzi na wysłaną dopiero co ankietę),
- należy starać się uzyskać w pierwszej kolejności potwierdzenia otrzymania fragmentów, aby w miarę możliwości wcześnie wykryć zagubione dane i tym samym zminimalizować ilość retransmitowanych danych,
- należy wcześniej pytać o status klientów, którzy bardziej zalegają z potwierdzaniem, aby uniknąć przestojów w wysyłaniu (czekając na ich potwierdzenia).

Liczba ankiet w pakiecie

Do każdego pakietu z danymi można dołączyć pewną, niewielką liczbę zapytań o status. Nie może to być zbyt duża liczba, gdyż spowodowałoby to lawinę odpowiedzi. Nie może to być też liczba zbyt mała, gdyż przy niezbyt dużym buforze kanału brak informacji o stanie klienta powodowałby niepotrzebne opóźnienia.

Wyznaczenie powyższej liczby składa sie z kilku etapów. Pierwszym jest wyznaczenie tzw. gęstości zapytań ($\rho_{poll(Fragment)}$). Jest to liczba klientów, którzy powinni odpowiedzieć na pytanie, na każdy wysłany bajt danego fragmetu, aby zapewnić maksymalną przepustowość, przy założeniach, że:

- dane nie są gubione,
- częstotliwości odpytywań wszystkich klientów zapisanych na dany strumień są równe,
- opóźnienia odpowiedzi na zapytania przez tych klientów są stałe,
- bufory nadawcze są maksymalnie wykorzystywane,
- klienci na bieżąco przetwarzają dane (nie ma przestojów).

$$\rho_{poll(Fragment)} = \frac{N_{clients(Fragment)}}{Channel_{BUFSIZE}}, gdzie$$

- $N_{clients(Fragment)}$ – liczba klientów zarejestrowanych na fragment,
- $Channel_{BUFSIZE}$ – rozmiar bufora kanału.

Po uwzględnieniu sugestii zawartej w [Lane], że odpytywanie powinno następować co około 80 - 90 procent wielkości bufora (przyjmijmy 85):

$$\rho_{poll(Fragment)} = \frac{N_{clients(Fragment)}}{Channel_{BUFSIZE} * 0,85}$$

Jeśli przyjąć, że liczba klientów, których należy zapytać o status przy wysyłaniu pakietu $Packet$ wynosi

$$N_{clients(Packet)} = \sum_{Fragment \in Packet} \rho_{poll(Fragment)} * Size(Fragment),$$

oraz, że można wysłać ankiety tylko do klientów, do których równocześnie wysyłamy fragmenty z danymi, to okazałoby się, że w przypadku zgubienia odpowiedzi nie byłoby możliwe ponowne wysłanie ankiety. W związku z tym liczbę ankietowanych klientów należy zwiększyć.

Zauważmy, że jeśli z danego kanału zaproponowany fragment nie jest wysyłany w tym pakiecie, to dlatego, że się nie zmieścił. W takiej sytuacji ankieta ma szansę być wysłana następnym razem.

Inaczej jest jeśli z danego kanału fragment nie został zaproponowany, ponieważ najpierw należy otrzymać odpowiedź od klienta. Wówczas należy dopuścić wysłanie do niego ankiety.

Proponuję zatem aby ostatecznie liczbę ankiet w pakiecie określał wzór

$$N_{clients(Packet)} = \sum_{Fragment \in Packet} \rho_{poll(Fragment)} * Size(Fragment) + \epsilon,$$

gdzie ϵ to np. suma logarytmów klientów zapisanych na kanały, które nie zaproponowały fragmentów, bo oczekują odpowiedzi. Z powyższego wzoru wynika także, że możliwe jest wysyłanie pakietów z ankietami, ale bez fragmentów.

Wyznaczenie klientów

Dla każdego kanału pamiętana jest kolejka priorytetowa klientów, których należy zapytać o status. Klienci są z niej usuwani przy wysyłaniu ankiety, natomiast dodawani podczas proponowania fragmentów do wysłania (patrz punkt 5.8.6). Priorytety zależą od czasu przebywania w kolejce (im dłużej tym wyższy) oraz od priorytetu początkowego zgłaszanego przy dodawaniu klienta do kolejki (im więcej ma danych do potwierdzenia, tym wyższy priorytet, przy czym brak potwierdzenia otrzymania ma wyższy priorytet niż brak potwierdzenia obsłużenia).

Dla każdego kanału pamiętany jest zbiór klientów, których stan nie zmienił się od czasu wysłania do nich ostatniej ankiety. Ma to na celu zabezpieczenie się przed wielokrotnym wysyłaniem ankiety, która nic nie wnosi. Klient dodawany jest do zbioru przy wysyłaniu do niego ankiety, natomiast usuwany po otrzymaniu zmianie listy fragmentów, które klient musi potwierdzić, czyli przy przetwarzaniu odpowiedzi oraz przy tworzeniu nowych fragmentów.

Klienci do ankietowania wyznaczani są za pomocą losowania w następujący sposób. Przeprowadzane jest $\ulcorner N_{clients(Packet)} \urcorner$ losowań kanałów. Każdy z kanałów ma prawdopodobieństwo wylosowania odpowiadające wkładowi do wspomnianej liczby. Po wylosowaniu kanału

wybierany i usuwany jest pierwszy klient z kolejki priorytetowej. Klient ten jest również usuwany z pozostałych kolejek. Ankieta jest do niego wysyłana, jeśli nie znajduje się w zbiorze klientów, których stan się nie zmienił, lub minęło już wystarczająco dużo czasu od wysłania do niego ostatniej ankiety (np. pół sekundy).

5.8.8. Obsługa odpowiedzi klientów

Po otrzymaniu odpowiedzi od klienta uaktualniany jest jego stan.

- Potwierdzane jest otrzymanie przez klienta wszystkich fragmentów należących do strumieni wcześniejszych niż $Fragment_{RCVD}.CS_{SEQ}$ (patrz punkt 5.7.2) oraz wszystkich fragmentów strumienia $Fragment_{RCVD}.CS_{SEQ}$ wcześniejszych niż $Fragment_{RCVD}.F_{SEQ}$. Odbywa się to poprzez przeniesienie klienta z list klientów, którzy jeszcze nie potwierdzili otrzymania fragmentów do odpowiadających im list klientów, którzy nie potwierdzili przetworzenia fragmentów.
- Potwierdzane jest przetworzenie przez klienta wszystkich fragmentów należących do strumieni wcześniejszych niż $Fragment_{RLSD}.CS_{SEQ}$ oraz wszystkich fragmentów strumienia $Fragment_{RLSD}.CS_{SEQ}$ wcześniejszych niż $Fragment_{RCVD}.F_{SEQ}$. Odbywa się to poprzez usunięcie klienta z list klientów, którzy jeszcze nie potwierdzili przetworzenia fragmentów. Najstarsze fragmenty, jeśli zostały już potwierdzone przez wszystkich klientów, są zwalniane.
- Uaktualniany jest numer ostatniego otrzymanego przez klienta pakietu.

5.8.9. Retransmisje

Retransmisja fragmentu następuje po stwierdzeniu, że istnieje klient, który otrzymał pakiet nowszy, lub dokładnie ten, w którym fragment ten został ostatnio wysłany, natomiast nie potwierdził on otrzymania tego fragmentu. Poza tym, retransmitowane fragmenty traktowane są identycznie jak fragmenty wysyłane po raz pierwszy. Szczegóły opisane są w punkcie 5.8.5. Przy retransmisji uaktualniany jest numer pakietu, w którym fragment ten został ostatnio wysłany.

5.9. Klient

Każde połączenie obsługiwane jest niezależnie, w związku z czym opisany zostanie sposób działania pojedynczego połączenia. Jedyną częścią wspólną może być, ale nie musi, identyfikator klienta ($Client_{Id}$).

5.9.1. Moduły

Klienta można podzielić na następujące moduły:

- interfejsu aplikacji klienckiej (IA).
- przetwarzający pakiety odebrane od serwera (PD),
- wysyłający status klienta (WS),

Moduł przetwarzający i wysyłający status jest jeden (dla połączenia), natomiast modułów interfejsu aplikacji wiele – po jednym dla każdego kanału.

5.9.2. Dane połączenia

Wspólnymi danymi wszystkich kanałów danego połączenia są jedynie identyfikator klienta oraz numer ostatnio otrzymanego pakietu.

Identyfikator klienta

Identyfikator klienta ($Client_{Id}$) jest stałą. Sposób jego nadawania nie jest tematem niniejszej pracy. Protokół nie zapewnia poprawnego działania w przypadku wystąpienia dwóch klientów o tym samym identyfikatorze, chociaż możliwa jest jego odpowiednia modyfikacja. Zaleca się, aby identyfikator klienta był taki sam dla poszczególnych połączeń. Jeśli na jednym komputerze pracuje nie więcej niż jedna aplikacja, to można przyjąć, że identyfikatorem klienta jest końcówka numeru IP tego komputera.

Numer ostatnio otrzymanego pakietu

Numer ostatnio ostatnio otrzymanego pakietu ($lastP_{SEQ}$) jest uaktualniany przez moduł PD po przetworzeniu pakietu. Służy mu jednocześnie do wykrywania pakietów przychodzących w innej kolejności niż zostały wysłane. Moduł WS wysyła do serwera $lastP_{SEQ}$ jako część statusu klienta.

5.9.3. Struktury danych kanału

Do poprawnego działania klienta potrzebne są struktury opisane w tym punkcie.

Bufor kanału

Na bufor kanału składa się liczba określająca jego maksymalny rozmiar ($Channel_{BUFSIZE}$) oraz lista fragmentów, które nie zostały jeszcze w całości przekazane aplikacji klienckiej ($Channel_{Fragments}$).

Fragmenty są umieszczane w buforze przez moduł PD. Może on też przeprowadzać operację „anulowania strumienia" (patrz punkt 5.9.7) oraz „odśmiecania", opisaną w punkcie 5.9.8, polegającą na usuwaniu fragmentów, które nie zostały przez klienta zamówione, w celu zwolnienia miejsca na nowe dane. Pozostałe fragmenty usuwane są wyłącznie przez moduł IA po przekazaniu ich aplikacji. Serwer dba o to, aby wszystkie zamówione fragmenty zmieściły się w buforze klienta (patrz punkt 5.8.6).

Fragmenty

Klient pamięta fragmenty w trochę innej postaci niż je serwer wysyła:

- numer strumienia według numeracji serwera (S_{SEQ}) – pozostaje bez zmian,
- numer fragmentu (F_{SEQ}) – pozostaje bez zmian,
- dane – przy wstawianiu do bufora tworzona jest ich kopia,
- znacznik końca strumienia (EOS) – pozostaje bez zmian,
- znacznik podsłuchania ($Snooped$) – ustawiany jeśli fragment został podsłuchany, w przeciwnym przypadku czyszczony – taki fragment możne zostać usunięty z bufora przez moduł PD w wyniku operacji anulowania lub odśmiecania,

- znacznik punktu synchronizacji ($SYNC$) – pozostaje bez zmian,
- numer strumienia wg numeracji klienta (CS_{SEQ}) – dla punktów bez znacznika $SYNC$ jest ustawiany tak jak wcześniejszy fragmentu tego samego strumienia (wartość kopiowana ze zmiennej $Current_{CS_{SEQ}}$), natomiast dla punktu ze znacznikiem $SYNC$ ustawiany tak, jak to określono w punkcie 5.9.6.

Najstarszy fragment

Najstarszy fragment ($Fragment_{OLDEST}$) to ten, który jest przekazywany aplikacji przez moduł IA, albo ten, o którym wiadomo, że będzie przekazany jako następny. Jest z nim związany wskaźnik ($Offset$) oddzielający dane pakietu przekazane aplikacji od pozostałych.

Numery bieżącego strumienia

Bieżący strumień to ten, z którego klient spodziewa się otrzymać następny fragment. Pamiętane są dwa numery:

- numer wg numeracji klienta ($Current_{CS_{SEQ}}$) – przesyłany tylko w punktach synchronizacji,
- numer wg numeracji serwera ($Current_{S_{SEQ}}$) – przesyłany w każdym fragmencie.

$Current_{CS_{SEQ}}$ wchodzi w skład statusu klienta.

Numer kolejnego fragmentu

Numer kolejnego fragmentu ($Next_{F_{SEQ}}$) jest numerem następnego fragmentu z bieżącego strumienia, którego klient spodziewa się dostać. Wchodzi on w skład statusu klienta.

Znacznik podsłuchiwania

Znacznik podsłuchiwania (*Snooping*) decyduje o tym czy bieżący strumień jest podsłuchiwany. Bieżący strumień jest podsłuchiwany, jeśli wszystkie fragmenty tego strumienia, przynajmniej od ostatniego punktu synchronizacji, zostały umieszczone w buforze (być może już przekazane aplikacji) i od tego czasu strumień ten nie został anulowany.

Znacznik ten nie może być ustawiony jednocześnie ze znacznikiem *Subscribed*.

Znacznik zapisania

Znacznik zapisania (*Subscribed*) decyduje o tym czy na bieżący strumień jest zapisany klient. Od klienta wymaga się potwierdzeń otrzymania i przetworzenia fragmentów, na które jest zapisany.

Znacznik ten nie może być ustawiony jednocześnie ze znacznikiem *Snooping*.

Status klienta

Status klienta jest wysyłany przez moduł WS do serwera. Jego zawartość podana jest w punkcie 5.7.2 opisującym zawartość pakietu z odpowiedzią klienta.

5.9.4. Przetwarzanie pakietów odebranych od serwera

Przetwarzaniem pakietów odebranych od serwera zajmuje się moduł PD.

Pierwszym krokiem jest porównanie numeru otrzymanego pakietu (P_{SEQ}) z otrzymanym poprzednio ($lastP_{SEQ}$). W tym celu obliczana jest

$$\Delta_{P_{SEQ}} = \begin{cases} P_{SEQ} - lastP_{SEQ} & \text{jeśli } P_{SEQ} - lastP_{SEQ} <= 0, \\ P_{SEQ} - lastP_{SEQ} & \text{jeśli } 0 < P_{SEQ} - lastP_{SEQ} < Max_{deltaPSeq}, \\ Max_{deltaPSeq} & \text{jeśli } Max_{deltaPSeq} < P_{SEQ} - lastP_{SEQ}, \end{cases}$$

gdzie $Max_{deltaPSeq}$ to mniejsza spośród maksymalnych wartości jakie mogą przyjmować S_{SEQ} i F_{SEQ} powiększonych o 1. Zakładając, że S_{SEQ} ma n bitów, a F_{SEQ} m bitów

$$Max_{deltaPSeq} = 2^{\min(m,n)}.$$

Jeśli $\Delta_{P_{SEQ}} <= 0$, to pakiet ten jest w całości pomijany. W przeciwnym przypadku przetwarzane są wszystkie fragmenty wydobyte z pakietu. Przetwarzanie zależy od tego czy bieżący strumień jest podsłuchiwany (*Snooped*) oraz od obecności znacznika punktu synchronizacji ($SYNC$). Fragmenty bez tego znacznika (w skrócie „fragmenty bez $SYNC$") można zidentyfikować jedynie za pomocą numeru strumienia S_{SEQ} oraz numeru fragmentu F_{SEQ}. Do identyfikacji fragmentów $SYNC$ można się dodatkowo posłużyć numerem strumienia klienta CS_{SEQ}.

Na końcu uaktualniana jest wartość $lastP_{SEQ}$ oraz sprawdzana obecność ankiety skierowanej do tego klienta. Jeśli taka istnieje, to informowany jest o tym moduł WS.

5.9.5. Przetwarzanie fragmentu bez znacznika $SYNC$

Pierwszym krokiem przetwarzania jest identyfikacja fragmentu, czyli sprawdzenie czy jest on starszy, taki sam, czy też nowszy niż spodziewany. Dopiero po tym można przystąpić do właściwego przetwarzania fragmentu.

Identyfikacja fragmentu

W celu identyfikacji fragmentu porównywany jest jego numeru strumienia S_{SEQ} z bieżącym numerem strumienia $Current_{S_{SEQ}}$ oraz numer fragmentu F_{SEQ} z bieżącym numerem kolejnego fragmentu $Next_{F_{SEQ}}$. Sposób i wynik porównywania zależy od tego czy w danej chwili podsłuchujemy (*Snooping*), czy obieramy dane, na które jesteśmy zapisani (*Subscribed*).

1. Przetwarzany fragment jest tym, którego się spodziewamy, czyli kolejnym z bieżącego strumienia, jeśli

$$Subscribed \wedge \left(S_{SEQ} = Current_{S_{SEQ}} \wedge F_{SEQ} = Next_{F_{SEQ}}\right)$$

albo

$$Snooping \wedge \left(\Delta_{P_{SEQ}} < Max_{deltaPSeq}\right) \wedge$$
$$\wedge \left(S_{SEQ} = Current_{S_{SEQ}} \wedge F_{SEQ} = Next_{F_{SEQ}}\right).$$

2. Przetwarzany fragment jest kolejnym fragmentem strumienia, którego ani nie podsłuchujemy, ani nie jesteśmy na niego zapisani, jeśli

$$\neg Subscribed \wedge \left(S_{SEQ} = Current_{S_{SEQ}} \wedge F_{SEQ} = Next_{F_{SEQ}}\right)$$

albo

$$\neg Snooping \wedge \left(\Delta_{P_{SEQ}} < Max_{deltaPSeq}\right) \wedge$$

$$\wedge \left(S_{SEQ} = Current_{S_{SEQ}} \wedge F_{SEQ} = Next_{F_{SEQ}}\right).$$

3. Przetwarzany fragment jest starszy niż się spodziewamy, czyli jest retransmitowany, jeśli

   $$Subscribed \wedge \left(\left(S_{SEQ} < Current_{S_{SEQ}}\right) \vee \left(S_{SEQ} = Current_{S_{SEQ}} \wedge F_{SEQ} < Next_{F_{SEQ}}\right)\right)$$

   albo

   $$\neg Subscribed \wedge \left(\Delta_{P_{SEQ}} < Max_{deltaPSeq}\right) \wedge$$

   $$\wedge \left(\left(S_{SEQ} < Current_{S_{SEQ}}\right) \vee \left(S_{SEQ} = Current_{S_{SEQ}} \wedge F_{SEQ} < Next_{F_{SEQ}}\right)\right).$$

4. Przetwarzany fragment jest nowszy niż się spodziewamy, co jest równoznaczne ze zgubieniem poprzednich fragmentów, jeśli

   $$Subscribed \wedge \left(\left(S_{SEQ} > Current_{S_{SEQ}}\right) \vee \left(S_{SEQ} = Current_{S_{SEQ}} \wedge F_{SEQ} > Next_{F_{SEQ}}\right)\right)$$

   albo

   $$\neg Subscribed \wedge \left(\Delta_{P_{SEQ}} < Max_{deltaPSeq}\right) \wedge$$

   $$\wedge \left(\left(S_{SEQ} > Current_{S_{SEQ}}\right) \vee \left(S_{SEQ} = Current_{S_{SEQ}} \wedge F_{SEQ} > Next_{F_{SEQ}}\right)\right).$$

5. Nic nie można powiedzieć o przetwarzanym fragmencie, jeśli

   $$\neg Subscribed \wedge \left(\Delta_{P_{SEQ}} = Max_{deltaPSeq}\right).$$

Zawsze możemy jednoznacznie zidentyfikować przetwarzany fragment, gdy jesteśmy zapisani (*Subscribed*), ponieważ serwer dba o to, aby nie wysłać zbyt dużo fragmentów lub strumieni bez potwierdzenia.

Przetwarzanie fragmentu retransmitowanego

Fragmenty retransmitowane są pomijane.

Przetwarzanie fragmentu podczas podsłuchiwania po zgubieniu poprzednich

Jeśli bieżący strumień jest podsłuchiwany (czyli nie był zamówiony) i podczas identyfikacji fragmentu okazało się, że jakiś fragment został zgubiony, to bieżący strumień jest anulowany, ponieważ nie można liczyć na to, że serwer prześle ponownie zagubione fragmenty. Uaktualniane są $Current_{S_{SEQ}}$ oraz $Next_{F_{SEQ}}$ po to, żeby zabezpieczyć się przed podsłuchiwaniem retransmitowanych fragmentów.

Jeśli fragment ma ustawiony znacznik *EOS*, to

- $Current_{S_{SEQ}} := S_{SEQ} + 1$,
- $Next_{F_{SEQ}} := 0$,
- *Snooping* jest wyłączany,

- *Subscribed* pozostaje wyłączony.

Jeśli fragment nie ma ustawionego znacznika EOS, to

- $Current_{S_{SEQ}} := S_{SEQ}$,
- $Next_{F_{SEQ}} := F_{SEQ} + 1$,
- *Snooping* pozostaje włączony,
- *Subscribed* pozostaje wyłączony.

Wartość $Current_{CS_{SEQ}}$ pozostaje bez zmian.

Przetwarzanie fragmentu nie zamówionego ani nie podsłuchiwanego

Fragment, który nie został zamówiony ani nie jest podsłuchiwany ($\neg Subscribed \wedge \neg Snooping$) jest zawsze pomijany. Jeśli nie jest retransmitowany, to uaktualniane są znaczniki:

- $Current_{S_{SEQ}} := \begin{cases} S_{SEQ} + 1 & \text{jeśli fragment ma znacznik } EOS, \\ S_{SEQ} & \text{w przeciwnym przypadku,} \end{cases}$
- $Next_{F_{SEQ}} := \begin{cases} 0 & \text{jeśli fragment ma znacznik } EOS, \\ F_{SEQ} + 1 & \text{w przeciwnym przypadku,} \end{cases}$
- *Snooping* pozostaje wyłączony,
- *Subscribed* pozostaje wyłączony.

Przetwarzanie niezidentyfikowanego fragmentu

Po napotkaniu fragmentu, o którym nic nie można powiedzieć, postępujemy podobnie jak w przypadku zagubienia poprzednich fragmentów – bieżący strumień jest anulowany (jeśli *Snooping*), $Current_{S_{SEQ}}$, $Next_{F_{SEQ}}$ są uaktualniane, a *Snooping* jest wyłączany (o ile już wcześniej nie był wyłączony).

W tym momencie powstaje niebezpieczeństwo, że niezidentyfikowany fragment tak naprawdę był retransmisją, a uaktualnienie bieżącego strumienia było błędem. Jako przykład niech posłuży następujący scenariusz:

1. Klient podsłuchuje strumień $S_{SEQ} = 1$ – dane są przekazywane na bieżąco aplikacji.
2. Serwer wysyła dużo pakietów (np. z danymi z innych kanałów), a klient gubi je wszystkie.
3. Serwer retransmituje od początku strumień $S_{SEQ} = 0$.
4. Klient gubi pierwszy fragment retransmitowanego strumienia, po czym przetwarza następny – bez znacznika $SYNC$.
5. Klient stwierdza, nie może zidetyfikować retransmitowanego pakietu, w związku z czym „cofa" bieżący strumień.
6. Serwer retransmituje kolejny fragment strumienia $S_{SEQ} = 0$, tym razem fragment ma ustawiony znacznik $SYNC$.

7. Klient zaczyna podsłuchiwać strumień $S_{SEQ} = 0$ i przekazuje dane aplikacji.

W ten sposób aplikacja otrzymała dane w nieprawidłowej kolejności (najpierw ze strumienia $S_{SEQ} = 1$, później $S_{SEQ} = 0$).

Prawdopodobieństwo wystąpienia powyższego scenariusza zależy od przyjętych maksymalnych wielkości F_{SEQ}, S_{SEQ}, szybkości sieci, prawdopodobieństwa zgubienia kolejnych $Max_{deltaPSeq}$ pakietów oraz przesyłania charakterystycznych strumieni i konieczności ich retransmitowania. Niebezpieczeństwo to można wyeliminować na przynajmniej dwa sposoby, z czego jeden nie wymaga żadnych zmian w protokole. Wystarczy:

- zapisywać F_{SEQ} oraz S_{SEQ} na odpowiednio dużej liczbie bitów, dzięki czemu zwiększa się $Max_{deltaPSeq}$, lub
- do każdego retransmitowanego fragmentu dodawać znacznik „retransmitowany", dzięki czemu zawsze będzie można zidentyfikować fragment (znacznik ten należy uwzględniać tylko w przypadku wątpliwości).

Przetwarzanie zamówionego fragmentu po zgubieniu poprzednich

Jeśli bieżący strumień jest zamówiony (*Subscribed*) i podczas identyfikacji fragmentu okaże się, że jakiś fragment został zgubiony, to fragment ten jest pomijany. Nie są uaktualniane zmienne $Current_{S_{SEQ}}$, $Current_{CS_{SEQ}}$, $Next_{F_{SEQ}}$, *Snooping* ani *Subscribed*, ponieważ spodziewamy się, że serwer ponownie wyśle zagubione fragmenty.

Przetwarzanie kolejnego zamówionego fragmentu

Przetworzenie kolejnego zamówionego fragmentu sprowadza się do:

1. Upewnienia się, że w buforze jest wystarczająca ilość wolnego miejsca. Jeśli nie ma, to bufor jest odśmiecany (patrz punkt 5.9.8). Jeśli po odśmieceniu bufora nadal nie ma miejsca, to oznacza to, że konfiguracja serwera i klienta nie pasują do siebie (parametr $Channel_{BUFSIZE}$ ma większą wartość na serwerze niż na kliencie).
2. Umieszczenia fragmentu w buforze.
3. Uaktualnienia znaczników.
4. Poinformowania modułu IA o nowych danych.

Jeśli fragment ma ustawiony znacznik EOS, oznaczający koniec strumienia, to znaczniki uaktualniane są następująco:

- $Current_{S_{SEQ}} := Current_{S_{SEQ}} + 1$ (równoznaczne $Current_{S_{SEQ}} := S_{SEQ}$),
- $Current_{CS_{SEQ}} := Current_{CS_{SEQ}} + 1$,
- $Next_{F_{SEQ}} := 0$,
- *Snooping* pozostaje wyłączony,
- *Subscribed* jest wyłączany, gdyż skończył się strumień,

W przeciwnym przypadku:

- $Current_{S_{SEQ}}$ pozostaje bez zmian,

- $Current_{CS_{SEQ}}$ pozostaje bez zmian,
- $Next_{F_{SEQ}} := Next_{F_{SEQ}} + 1$, co jest równoznaczne $Next_{F_{SEQ}} := F_{SEQ}$.
- *Snooping* pozostaje wyłączony,
- *Subscribed* pozostaje włączony.

Przetwarzanie kolejnego podsłuchiwanego fragmentu

Przetworzenie kolejnego podsłuchiwanego fragmentu rozpoczyna się od próby umieszczenia fragmentu w buforze.

Jeśli nie ma wystarczającej ilości wolnego miejsca, to bieżący strumień jest anulowany (patrz punkt 5.9.7), po czym uaktualniane są struktury kanału w identyczny sposób, jak w przypadku zagubienia podsłuchiwanych fragmentów.

Jeśli natomiast jest miejsce w buforze to:

1. Fragment umieszczany jest w buforze,
2. Uaktualniane są struktury:
   - $Current_{S_{SEQ}} := \begin{cases} S_{SEQ} + 1 & \text{jeśli fragment ma znacznik } EOS, \\ S_{SEQ} & \text{w przeciwnym przypadku,} \end{cases}$
   - $Next_{F_{SEQ}} := \begin{cases} 0 & \text{jeśli fragment ma znacznik } EOS, \\ F_{SEQ} + 1 & \text{w przeciwnym przypadku,} \end{cases}$
   - $Snooping \begin{cases} \text{jest wyłączany} & \text{jeśli fragment ma znacznik } EOS, \\ \text{pozostaje włączony} & \text{w przeciwnym przypadku,} \end{cases}$
   - *Subscribed* pozostaje wyłączony.
3. Moduł IA informowany jest o nowych danych.

5.9.6. Przetwarzanie fragmentu ze znacznikiem $SYNC$

Przetwarzanie fragmentu ze znacznikiem $SYNC$, podobnie jak w przypadku fragmentu bez znacznika, składa się z dwóch kroków. Pierwszym jest identyfikacja fragmentu, a drugim właściwe przetworzenie go.

Identyfikacja fragmentu

Identyfikacja fragmentu zaczyna się sprawdzenia czy ten klient ($Client_{Id}$) znajduje się na liście starych ($Clients_{OLD}$), albo nowych ($Clients_{NEW}$) klientów, którzy zamówili ten fragment. Obecność na jednej z nich oznacza, że klient jest na ten fragment zapisany. Przy okazji ustalany jest numer strumienia według numeracji klienta (CS_{SEQ}), do którego należy przetwarzany fragment.

$$CS_{SEQ} := \begin{cases} CS_{SEQ}(Clients_{NEW}(Client_{Id})), & \text{jeśli } Client_{Id} \in Clients_{NEW}, \\ CS_{SEQ}(Clients_{OLD}(Client_{Id})), & \text{jeśli } Client_{Id} \in Clients_{OLD}, \\ CS_{SEQ}(Clients_{KNOWN}(Client_{Id})), & \text{jeśli } Client_{Id} \in Clients_{KNOWN}, \\ 0 & \text{w przeciwnym przypadku.} \end{cases}$$

$$Client_{NEW} := Client_{Id} \in Clients_{NEW}$$
$$Client_{OLD} := Client_{Id} \in Clients_{OLD}$$

Identyfikacja fragmentu ze znacznikiem $SYNC$ przebiega w sposób zbliżony do identyfikacji fragmentu bez $SYNC$. Poniżej podano najważniejsze zmiany.

1. Warunkiem wystarczającym do stwierdzenia retransmisji fragmentu jest
$$CS_{SEQ} < Current_{CS_{SEQ}}$$
oraz
$$Subscribed \wedge \left(CS_{SEQ} = Current_{CS_{SEQ}}\right) \wedge \left(F_{SEQ} < Next_{F_{SEQ}}\right).$$

2. Warunkiem wystarczającym do stwierdzenia zgubienia zamówionego fragmentu, co oznacza konieczność całkowitego pominięcia przetwarzanego, jest
$$CS_{SEQ} > Current_{CS_{SEQ}}$$
oraz
$$Subscribed \wedge \left(CS_{SEQ} = Current_{CS_{SEQ}}\right) \wedge \left(F_{SEQ} > Next_{F_{SEQ}}\right).$$

3. Warunkiem koniecznym do rozpoczęcia przetwarzania fragmentu jest
$$CS_{SEQ} = Current_{CS_{SEQ}}.$$

4. Przetwarzany fragment jest pierwszym zamówionym, czyli należy go zacząć przetwarzać, jeśli
$$Client_{NEW} \wedge \left(CS_{SEQ} = Current_{CS_{SEQ}}\right).$$
Dodatkowo, jeśli spełniony jest warunek
$$Snooping \wedge \left(\Delta_{P_{SEQ}} < Max_{deltaPSeq}\right) \wedge$$
$$\wedge \left(S_{SEQ} = Current_{S_{SEQ}} \wedge F_{SEQ} = Next_{F_{SEQ}}\right),$$
to można kontynuować przetwarzanie podsłuchiwanego strumienia. W przeciwnym przypadku należy go anulować (o ile podsłuchiwano).

5. Przetwarzany fragment jest kolejnym zamówionym z bieżącego strumienia, jeśli
$$Client_{OLD} \wedge \left(CS_{SEQ} = Current_{CS_{SEQ}} \wedge F_{SEQ} = Next_{F_{SEQ}}\right).$$
Zauważmy, że tym przypadku dodatkowo powinno zachodzić
$$Subscribed \wedge \left(S_{SEQ} = Current_{S_{SEQ}}\right).$$

6. Przetwarzany fragment jest kolejnym podsłuchiwanym z bieżącego strumienia, jeśli
$$(\neg Client_{OLD} \wedge \neg Client_{NEW}) \wedge Snooping \wedge \left(\Delta_{P_{SEQ}} < Max_{deltaPSeq}\right) \wedge$$
$$\wedge \left(CS_{SEQ} = Current_{CS_{SEQ}} \wedge S_{SEQ} = Current_{S_{SEQ}} \wedge F_{SEQ} = Next_{F_{SEQ}}\right).$$

7. Przetwarzany fragment jest nowszy niż się spodziewamy i można go zacząć podsłuchiwać po anulowaniu bieżącego strumienia, jeśli
$$(\neg Client_{OLD} \wedge \neg Client_{NEW}) \wedge$$
$$\wedge \left(CS_{SEQ} = Current_{CS_{SEQ}}\right) \wedge$$
$$\wedge \left(\left(S_{SEQ} > Current_{S_{SEQ}}\right) \vee \left(S_{SEQ} = Current_{S_{SEQ}} \wedge F_{SEQ} > Next_{F_{SEQ}}\right)\right).$$
Zauważmy, że znacznik $Subscribed$ nie powinien być ustawiony.

Rozpoczęcie podsłuchiwania strumienia

Rozpoczęcie podsłuchiwania strumienia sprowadza się do ustawienia znacznika *Snooping*, oraz zaznaczenia, że przetwarzany fragment jest jego początkiem. Dalej należy postępować identycznie jak w przypadku kontynuowania podsłuchiwania.

Rozpoczęcie przetwarzania zamówionego strumienia

Przy rozpoczynaniu przetwarzania zamówionego strumienia należy ustawić znacznik *Subscribed*, zaznaczyć, że przetwarzany fragment jest jego początkiem, po czym postępować analogicznie jak w przypadku przetwarzania kolejnego zamówionego fragmentu.

5.9.7. Anulowanie strumienia

Anulować można tylko podsłuchiwane strumienie. Operacja ta polega na usunięciu z bufora wszystkich fragmentów danego strumienia oraz poinformowaniu o tym modułu IA. Możliwe jest też anulowanie tylko części strumienia – od początku do fragmentu poprzedzającego punkt synchronizacji.

Skutkiem anulowania dla modułu PD jest potencjalne zwiększenie ilości wolnego miejsca w buforze. Ponadto, jeśli anulowany strumień jest strumieniem bieżącym oraz anulowano go w całości, to wyłączany jest znacznik *Snooping*.

Najbliższa próba pobrania danych z tego strumienia przez aplikację zakończy się wyjątkiem.

5.9.8. Odśmiecanie bufora

Odśmiecanie bufora polega na anulowaniu części lub wszystkich podsłuchanych fragmentów, które znadują się w buforze. Celem jest zwolnienie zadanej ilości miejsca w buforze, aby mógł się w nim zmieścić zamówiony fragment. Trudno jest określić optymalną kolejność anulowania strumieni. Wydaje się jednak, że jeśli w danym momencie aplikacja pobiera jakiś podsłuchany strumień, to powinien on być anulowany w ostatniej kolejności.

5.9.9. Przekazanie danych aplikacji

Dane są przekazywane aplikacji na jej prośbę – wykonanie funkcji odczytu danych ze strumienia. Moduł interfejsu aplikacji przekazuje dane z fragmentów zgromadzonych w buforze. W przypadku stwierdzenia anulowania strumienia, zamiast danych, przekazuje wyjątek niespodziewanego ich końca.

Fragment, po wykorzystaniu go, zostaje usunięty z bufora, o czym informowany jest moduł WS w celu przekazania serwerowi potwierdzenia.

5.9.10. Wysyłanie informacji o statusie

Wysyłaniem informacji o statusie klienta zajmuje się moduł WS. Reaguje on na przesłane ankiety, przekazane przez moduł PD, oraz informacje od modułu IA o przekazaniu danych aplikacji.

Odpowiedź może zostać wysłana w następujących okolicznościach:

1. Serwer przysłał ankietę.
2. Od czasu wysłania ostatniej odpowiedzi, aplikacji została przekazana duża ilość zamówionych danych. Dużą ilością jest np. $80\% * Channel_{BUFSIZE}$.

3. Od przekazania aplikacji wszystkich zamówionych fragmentów minęło trochę czasu i nie wysłano w tym czasie żadnej odpowiedzi. Trochę czasu może oznaczać np. średni czas potrzebny do przesłania danych o rozmiarze $2 * Channel_{BUFSIZE}$.

5.10. API serwera

Interfejs programistyczny serwera składa się dwóch interfejsów oraz jednej klasy abstrakcyjnej.

5.10.1. Publisher

Za pomocą interfejsu *Publisher* aplikacja uzyskuje dostęp do kanałów. Mogę one pochodzić z jednego lub wielu połączeń. Implementacja interfejsu powinna pozwalać na wybór strategii przydzielania kanałom połączeń, np. przez udostępnienie odpowiedniego kontruktora.

Interfejs składa się z następujących metod:

- **PublChannel channel(ChannelId channelId) throws IOException** – tworzy nowy kanał albo przekazuje istniejący o podanym identyfikatorze. Jeśli jest taka potrzeba, to tworzy nowe połączenie.
- **void close() throws IOException** – zamyka wszystkie kanały oraz wykorzystywane przez nie połączenia.

5.10.2. PublChannel

PublChannel reprezentuje jeden kanał. Najważniejszą jego metodą jest

- **PublStream createStream() throws IllegalStateException**,

która tworzy nowy strumień. Wspomniany wyjątek zgłaszany jest w sytuacji, gdy poprzedni strumień nie został jeszcze zamknięty.

Pozostałe metody pełnią funkcje informacyjne:

- **Publisher getPublisher()** – przekazuje obiekt *Publisher*a.
- **ChannelId getChannelId()** – przekazuje identyfikator kanału.
- **PublStream getCurrentStream()** – przekazuje ostatnio utworzony strumień, o ile nie został on jeszcze zamknięty.

5.10.3. PublStream

PublStream jest klasą abstrakcyjną rozszerzającą z *OutputStream*. Najważnejszymi dziedziczonymi metodami są:

- **void write(byte[] b, int off, int len) throws IOException** – zapisuje dane do strumienia, czyli rozgłasza je, oraz
- **void close() throws IOException** – zamykająca strumień.

Klasa *PublStream* wprowadza oczywiście nowe metody. Część z nich służy do pobierania informacji o stanie strumienia. Nie będę ich jednak tutaj przytaczał. Bardziej interesujące są następujące metody:

- **void subscribe(ClientId clientId)** – zapisuje klienta o podanym identyfikatorze na wysyłane dane począwszy od najbliższego punktu synchornizacji.
- **void sync()** – tworzy punkt synchronizacji.

Przykład

Kod wysłający dane nowym strumieniem do klientów cl1 i cl2, a od kolejnego punktu synchronizacji dodatkowo do klienta cl3 (pominięto obsługę wyjątków):

```
PublStream ps = publChannel.createSteam();
ps.subscribe(cl1);
ps.subscribe(cl2);
ps.write(data1);
ps.sync();
ps.subscribe(cl3);
ps.write(data2);
ps.close();
```

5.11. API klienta

Interfejs programistyczny klienta, podobnie jak serwera, składa się z dwóch interfejsów oraz jednej klasy abstrakcyjnej.

5.12. Subscriber

Subscriber jest interfejsem, który udostępnia kanały. Mogę one pochodzić z jednego lub wielu połączeń. Implementacja interfejsu klienta powinna pasować do implementacji interfejsu serwera, między innymi w kwestii przydzielania kanałom połączeń.

Interfejs składa się z takich samych metod jak *Publisher*. Różnią się one jedynie klasą przekazywanego strumienia.

- **SubscrChannel channel(ChannelId channelId) throws IOException** – tworzy nowy kanał albo zwraca istniejący o podanym identyfikatorze. Jeśli jest taka potrzeba to tworzy nowe połączenie.
- **void close() throws IOException** – zamyka wszystkie kanały oraz *Subscriber*a.

5.12.1. SubscrChannel

SubscrChannel reprezentuje jeden kanał. Najważniejszą jego metodą jest

- **SubscrStream nextStream(boolean subscribed, long milis) throws IllegalStateException, InterruptedException**,

która pobiera kolejny strumień. Jeśli poprzedni strumień nie został jeszcze zamknięty, to zgłaszany jest pierwszy podany wyjątek. Jeśli strumień jest dostępny od razu, to go przekazuje, w przeciwnym przypadku czeka co najwyżej podaną ilość czasu. Parametr *subscribed* decyduje o tym, czy przekazywany strumień może być podsłuchiwany (wówczas istnieje niebezpieczeństwo jego nagłego przerwania), czy też musi być zamówiony.

Pozostałe metody pełnią funkcje informacyjne:

- **Subscriber getSubscriber()** – przekazuje obiekt *Subscriber*a.
- **ChannelId getChannelId()** – przekazuje identyfikator kanału.
- **SubscrStream getCurrentStream()** – zwraca strumień, który ostatnio został dostarczony przez metodę *nextStream*, o ile nie został on jeszcze zamknięty.

5.12.2. SubscrStream

SubscrStream jest klasą abstrakcyjną rozszerzającą standardową klasę *InputStream*. Najważnejszymi dziedziczonymi metodami są:

- **int read(byte[] b, int off, int len) throws IOException** – odczytująca dane ze strumienia oraz
- **void close() throws IOException** – zamykająca strumień.

Klasa *SubscrStream* wprowadza oczywiście nowe metody. Część z nich służy do pobierania informacji o stanie strumienia. Najciekawszą z nich jest

- **boolean isSubscribed()**,

dzięki której można się dowiedzieć czy klient jest na strumień zapisany, czy tylko go podsłuchuje.

Przykład

Kod odczytujący zamówione dane z kolejnego strumienia (pominięto obsługę wyjątków):

```
SubscrStream ss = subscrChannel.nextSteam(true, 5000); // czekamy <= 5 sekund
if (ss != null) {
    byte[] buf = new byte[5000];
    int cnt;
    while ((cnt = ss.read(buf)) >= 0) {
        // zrobienie czegoś z danymi...
        ...
    }
    ss.close();
}
```

Rozdział 6

Implementacja

Implementacja protokołu, będącego tematem niniejszej pracy, powstawała równolegle z samym protokołem. Zaczęło się od prototypów, które miały za zadanie sprawdzenie działania bibliotek, służących do rozgłaszania i rozsyłania grupowego w systemach operacyjnych Windows i Linux. Miały też na celu sprawdzenie założeń teoretycznych dotyczących gubienia pakietów oraz zachowania całego protokołu.

Implementacja ta jest rozwinięciem wspomnianych prototypów i zawiera szereg uproszczeń, co sprawia, że ją także należy traktować jako prototyp, a nie jako gotowe rozwiązanie, które można by było zastosować w praktyce.

6.1. Java

Protokół został zaimplementowany w Javie. O motywach takiej decyzji można się dowiedzieć z punktu 3.3. Dodatkowymi argumentami przemawiającymi za wyborem Javy są jej zalety:

- brak błędów naruszenia ochrony pamięci,
- szybkość i łatwość znajdowania oraz usuwania błędów (dzięki łańcuchom wyjątków (ang. *exception chaining*),
- obsługa wątków i ich sychronizacji,
- bogate standardowe biblioteki,
- przenośność na różne systemy operacyjne,
- znakomite, darmowe środowisko programistyczne *Eclipse*.

Pierwsze prototypy wykorzystywały wersję 1.4.1, która szybko została zastąpiona kolejną – 1.4.2. W chwili ukazania się pierwszej, publicznie dostępnej, testowej wersji 1.5.0 zacząłem jej używać. Decydującymi argumentami za taką zmianą były:

- włączenie pakietu *java.util.concurrent* znacznie ułatwiającego pisanie aplikacji wielowątkowych, dzięki czemu mogłem zrezygnować między innymi z własnej implementacji semafora,
- generyczne kolekcje,
- nowa pętla *for* zwiększająca czytelność i szybkość pisania kodu.

6.2. Biblioteki

W skład Javy wchodzi bardzo dużo bibliotek. Podczas implementacji okazało się, że w większości są one wystarczające. Dlatego też potrzeby wykorzystania zewnętrzych pakietów okazały się niewielkie. Początkowo wykorzystywane były dwie zewnętrzne biblioteki. Docelowo można z obu zrezygnować.

util.concurrent

Pierwsze prototypy wykorzystywały bibliotekę *EDU.oswego.cs.dl.util.concurrent* autorstwa Doug Lea, będącą prekursorem pakietu *java.util.concurrent* ([JSR166]). W jej skład wchodzą implementacje różnych mechanizmów służących do synchronizacji i komunikacji wątków, takie jak semafory, zatrzaski (ang. *latch*), bariery (punkty sychronizacji wątków), zmienne z możliwością atomowego przeprowadzania na nich operacji oraz synchronizowane kolekcje. W momencie pojawienia się wersji 1.5.0 Javy można było zrezygnować z użycia tej biblioteki, gdyż większość jej klas została przeniesiona do pakietu *java.util.concurrent*. Obecnie z całej tej biblioteki implementacja wykorzystuje jedynie semafor o bardzo dużych możliwościach oraz wydajny synchronizowany słownik (*ConcurrentHashMap*).

org.apache.commons.lang

Drugą wykorzystywaną biblioteką jest *org.apache.commons.lang*. Szczególnie przydatna, głównie do odnajdywania i usuwania błędów, okazała się klasa *ToStringBuilder*, dzięki której implementacja podstawowej metody *toString()* sprowadza się do napisania jednego wiersza kodu. Do klasy tej wprowadzono małą modyfikację służącą poprawieniu czytelności generowanej przez nią tekstu.

Kolejna wykorzystana klasa to *Enum*, na której bazuje stan strumienia. Jedną z nowości w wersji 1.5.0 Javy są enumeracje, które służą do dokładnie tych samych celów co ta klasa. Nie zostały one jednak wykorzystane, gdyż kompilator środowiska programistycznego *Eclipse* w chwili pisania implementacji jeszcze ich nie obsługiwał.

6.3. Protokoły niższych warstw

Java niestety nie oferuje możliwości wysyłania i odbierania ramek ethernetowych ani pakietów IP. Jedynym możliwym rozwiązaniem okazała się implementacja bazująca na protokole UDP.

Do rozsyłania danych przez serwer oraz odbioru rozsyłanych danych przez klientów została wykorzystana klasa *MulticastSocket*, gdyż jej odpowiednik oparty o kanały (*MulticastChannel*) jest dopiero we wczesnej fazie specyfikacji ([JSR203]).

Klienci do wysyłania informacji o swoim stanie, a serwer do ich odbioru wykorzystuje klasy *DatagramSocket* oraz *DatagramChannel*, ale tylko w synchronicznym trybie pracy. Praca asynchroniczna okazała się trudna w realizacji, ze względu na różnice w działaniu na różnych systemach operacyjnych.

6.4. Konfiguracja

Konfiguracja serwera i klienta polega na przypisaniu tych samych wartości tym samym parametrom.

Część parametrów służy do konfiguracji działania samego protokołu i decyduje o wydajności i wymaganiach pamięciowych. W prototypie w celu ich zmiany konieczna jest zmiana stałych, a następnie kompilacja. W przypadku ustawienia różnych wartości na serwerze i kliencie protokół może przestać działać poprawnie.

- $Max_{Packet_{SIZE}}$ decyduje o maksymalnym rozmiarze wysyłanego pakietu. Serwer nie wysyła pakietów większych niż wskazywana wartość, a klient na jego podstawie ustala rozmiar bufora przeznaczonego do ich odbierania.

- $Channel_{BUFSIZE}$ decyduje o rozmiarze bufora kanału. Szczegóły na jego temat można znaleźć w rozdziale 5.

Pozostałe parametry nie mają żadnego wpływu na wydajność:

- adres oraz port rozsyłania grupowego,

- port, na którym serwer nasłuchuje odpowiedzi od klientów.

Ciekawostką jest to, że klient nie musi znać prawdziwego adresu serwera, chociaż musi do niego wysyłać odpowiedzi. Jest to możliwe dlatego, że pakiet wysyłany przez serwer zawiera jego adres, a klient nie musi odpowiadać przed jego otrzymaniem.

6.5. Serwer

Implementacja serwera dość wiernie odpowiada opisowi protokołu. Jedyne odstępstwa dotyczą przyjętych uproszczeń oraz rzeczy, które nie zostały w wystarczającym stopniu sprecyzowane.

Wątki

Każde połączenie ma dwa wątki:

- Główny. Analizuje odpowiedzi klientów, przetwarza dane z buforów kanałów na fragmenty, decyduje o kolejności i czasie ich wysłania, tworzy ankiety oraz buduje i rozsyła pakiety.

- Odbierający statusy klientów. Zajmuje się wyłączenie odbieraniem pakietów z odpowiedziami klientów oraz przekazywaniem ich wątkowi głównemu. Wątek główny budzony jest przez podniesienie semafora, na którym oczekuje.

Wątek główny po wykonaniu jedej sekwencji zadań zasypia na semaforze na pewien czas, którego długość zależy od wykonanych czynności.

Nie jestem przekonany, czy w tej chwili możliwe jest zrezygnowanie z drugiego wątku tak, aby jego zadania przejął wątek główny, bez wykorzystania dodatkowych bibliotek. Wydaje się, że będzie to możliwe dopiero po wprowadzeniu kanałów do rozgłaszania grupowego.

Bufory

Z każdym kanałem związany jest jeden bufor – taki, jak opisany w punkcie 5.8.3. Konieczne było zaimplementowanie mechanizmu synchronizacji, który powoduje, że wątek wstawiający dane do bufora jest blokowany do czasu zwolnienia w nim miejsca.

Kopiowanie

Każdy wysyłany bajt jest przed wysłaniem kopiowany przynajmniej trzy razy:

- przy wstawianiu danych do bufora kanału,
- przy budowaniu pakietu – kopiowany z bufora kanału do bufora pakietu,
- przy wysyłaniu – kopiowany z bufora pakietu do bufora gniazda.

Pierwsze kopiowanie można by było wyeliminować na dwa sposoby:

- rezygnując z bufora kanału, co sprowadzałoby się w skrajnych przypadkach do wysyłania po jednym bajcie w pakiecie albo
- zakazując klientowi powtórnego wykorzystania tablicy, w której przekazał dane, co prawdopodobnie znacznie utrudniłoby zastosowanie w praktyce.

Drugie oraz trzecie kopiowanie można wyeliminować dopiero po wprowadzeniu kanałów do rozsyłania grupowego. Rezygnacja z kopiowania przy budowaniu pakietu wymaga zastosowania funkcji interfejsu *GatheringByteChannel*, natomiast kopiowanie do bufora gniazda da się wyeliminować stosując bufor bezpośredni (ang. *direct buffer*).

6.6. Klient

Implementacja klienta, podobnie jak serwera, jest wierna opisowi protokołu. W niej również zostały przyjęte pewne uproszczenia.

Implementacja serwera dość wiernie odpowiada opisowi protokołu. Jedyne odstępstwa dotyczą przyjętych uproszczeń oraz rzeczy, które nie zostały w wystarczającym stopniu sprecyzowane.

Wątki

Każde połączenie ma dwa wątki:

- Główny. Odbiera pakiety wysłane przez serwer, przetwarza je, informuje wątki aplikacji o nowych danych oraz wznawia wątek wysyłający odpowiedzi, o ile jest taka potrzeba. Odpowiada modułowi PD z opisu protokołu.
- Wysyłający informacje o statusie. Zajmuje się wyłączenie wysyłaniem informacji o statusie do serwera.

Możliwe jest rozszerzenie funkcjonalności głównego wątku o wysyłanie informacji o statusie i tym samym wyeliminowanie drugiego wątku. Wymaga to jednak wprowadzenia asynchronicznej obsługi gniazd. Innym sposobem zmniejszenia liczby wątków jest wykorzystanie zarówno wątku głównego, jak i wątków aplikacji do wysyłania potwierdzeń.

Bufory

Z każdym kanałem związany jest jeden bufor – taki, jak opisany w punkcie 5.8.3. Nie został on jednak zaimplementowany jako standardowy bufor cykliczny, czyli tablica o określonej wielkości ze wskaźnikami pozycji zapisu i odczytu. Zamiast tego dane z poszczególnych fragmentów przechowywane są w oddzielnie zaalokowanych tablicach. Ponadto pamiętany jest stopień wykorzystania bufora, czyli liczba bajtów zajmowanych przez wszystkie te tablice.

Taka a nie inna budowa bufora wynika z konieczności przeprowadzania operacji „odśmiecania". Dzięki temu udało się uniknąć problemu jego fragmentacji – ciężar zarządzania pamięcią spoczywa na barkach maszyny wirtualnej.

Kopiowania

Każdy odebrany bajt jest kopiowany przynajmniej trzy razy:

- przy odbieraniu – kopiowany z bufora gniazda do bufora pakietu,
- przy przetwarzaniu – kopiowany z bufora pakietu do bufora kanału,
- przy przekazywaniu aplikacji – kopiowany z bufora kanału do tablicy przekazanej przez aplikację.

Pierwsze kopiowanie można wyeliminować dopiero po wprowadzeniu kanałów do rozsyłania grupowego, gdyż wymaga to zastosowania bufora bezpośredniego. Rezygnacja z drugiego kopiowania wymagałaby ograniczenia funkcjonalności protokołu – rezygnację z kanałów oraz strumieni (połączenie miałoby ściśle określoną, stałą liczbę klientów i umożliwiałoby przesłanie tylko jednego strumienia). Zmiana API polegająca na zmuszeniu aplikacji do wykorzystania danych znajdujących się w buforze dostarczonym przez implementację protokołu umożliwiłaby wyeliminowanie trzeciego kopiowania.

Jak widać jedynie ostatnia zmiana byłaby możliwa. Wydaje się jednak, że w chwili obecnej zysk wydajności nie usprawiedliwia przyjęcia mniej wygodnego w użyciu API.

6.7. Uproszczenia

Uproszczenia, które zostały przyjęte, dotyczą jedynie implementacji, a nie protokołu. Jestem przekonany, że docelowa, pełna implementacja, może być tylko lepsza i sprawniejsza.

Najważniejsze uproszczenia:

1. Przyjęcie, że numery P_{SEQ}, S_{SEQ}, CS_{SEQ}, F_{SEQ}, $Client_{Id}$ oraz $Channel_{Id}$ są zapisane na 32 bitach. Powoduje to, że w każdym pakiecie mniej miejsca pozostaje na dane. Na wszystkie powyższe numery, z wyjątkiem P_{SEQ} można by przeznaczyć dwukrotnie mniej miejsca.

2. Brak obsługi przekroczenia zakresów wszystkich numerów. W związku z tym nie jest możliwe przesłanie więcej niż około miliarda pakietów, strumieni, czy fragmentów w strumieniu.

3. Brak odporności na awarie klientów. Jeśli klient ulegnie awarii w momencie, gdy nie potwierdził wszystkich zamówionych danych, to serwer będzie cały czas wysyłał do niego co jakiś czas ankiety. Możliwym rozwiązaniem jest odłączanie kłopotliwego klienta w przypadku stwierdzenia braku jego odpowiedzi przez pewien czas.

4. Brak odporności na awarie serwera. Jeśli dojdzie do awarii serwera, to klienci spodziewający się danych będą na nie czekać w nieskończoność. Należy sprawdzać czas, jaki upłynął od ostatniej transmisji danych przez serwer lub wprowadzić inny mechanizm służący do wykrywania jego poprawnego działania.

5. Brak mechanizmu zarządzającego numerami klientów. W implementacji identyfikatory nadawane klientom nie są w żaden sposób weryfikowane. W szczególności nie jest sprawdzane czy dwóch klientów nie ma takiego samego. Docelowo musi powstać mechanizm zarządzający tymi identyfikatorami, a serwer, przy odbieraniu potwierdzeń od klientów, musi sprawdzać zgodność nadawcy z przekazanym przez niego identyfikatorem lub polegać wyłącznie na jego adresie.

6. Brak możliwości zamykania strumieni przez klientów. Od klientów wymaga się odbierania wszystkich zamówionych danych. Zamknięcie strumienia przez klienta można rozwiązać rozszerzając pakiet z odpowiedzią o informację o zamkniętych strumieniach.

7. Brak możliwości zamykania kanałów. Zamykanie kanałów nie zostało zaimplementowane.

Rozdział 7

Przykład zastosowania

Wielokrotnie spotykałem się z problemami, których jednym z możliwych rozwiązań było zastosowanie protokołu niezawodnego rozgłaszania.

Jednym z nich była konieczność częstego i szybkiego aktualizowania identycznych danych znajdujących się na kilkunastu urządzeniach połączonych wolną siecią z komputerem głównym. W chwili obecnej dane są przesyłane do każdego z urządzeń osobno, jednak prędkość, z jaką dane są aktualizowane, nie jest zadowalająca. Poprawić sytuację można jedynie przez wymianę urządzeń i rozbudowę sieci, co jest kosztowne, albo poprzez wykorzystanie protokołu niezawodnego rozgłaszania.

W tym rozdziale chcę jednak przedstawić inne zastosowanie.

7.1. Problem

Pisanie aplikacji w Javie ma wiele zalet, o których już była mowa we wcześniejszych rozdziałach. Niestety ma też wady. Jedna z nich związana jest z postacią, jaką przyjmuje skompilowana aplikacja.

Wynikiem kompilacji programu jest wiele plików – przynajmniej tyle, ile klas zawierają źródła. Modyfikacje fragmentu aplikacji często wpływają na zawartość jedynie części utworzonych plików. Czasami jednak jedna drobna zmiana wpływa na treść większości plików.

Pewna aplikacja po skompilowaniu składa się z kilkunastu tysięcy plików o łącznym rozmiarze ponad pięćdziesięciu megabajtów. Podczas jej wdrożenia konieczne było częste dokonywanie zmian i uzupełnień. W szczytowym okresie kilka razy w tygodniu wykonywane były mniejsze aktualizacje wpływające na zmianę od kilkunastu do kilkuset plików. Większe natomiast były wykonywane średnio co dwa - trzy tygodnie i wiązały się ze zmianą od kilkuset do kilku tysięcy plików.

Podczas wspomnianego wdrożenia aplikacja była zainstalowana jedynie na dwóch komputerach, dlatego nie było większych problemów z przeprowadzaniem aktualizacji. Wszystkie potrzebne pliki przegrywane były na dyski lokalne komputerów, skąd aplikacja była uruchamiana.

Kolejne wdrożenia będą się wiązać z koniecznością instalacji na wielu (od kilku do kilkudziesięciu) komputerach. Faktyczna przepustowość sieci u różnych klientów jest różna i wynosi od jednego do dziesięciu megabitów. Nadal będzie potrzeba przeprowadzania aktualizacji, chociaż nie tak częstych jak wcześniej. Rozmiar aplikacji oraz częstotliwość i zakres jej aktualizacji sprawia, że problemem staje się przeprowadzanie instalacji oraz aktualizacji.

7.2. Możliwe rozwiązania

Aplikację można zainstalować na wiele sposobów. Każdy z nich ma wady i zalety.

7.2.1. Instalacja na lokalnych dyskach komputerów

Najprostszym sposobem instalacji jest przegranie plików aplikacji na lokalne dyski komputerów. Niewątpliwą zaletą takiego rozwiązania jest to, że sieć nie jest obciążana przy uruchamianiu aplikacji. Potrzebne pliki (klasy) są wczytywane bezpośrednio z dysku komputera.

Rozwiązanie to niestety nie sprawdza się, gdy aplikacja działa na większej liczbie komputerów. Dzieje się tak dlatego, że każda aktualizacja pociąga za sobą konieczność przegrania plików na wszystkie komputery. Wymaga to czasu, którego jest zwykle za mało, oraz konieczności pozostawienia w tym czasie włączonych komputerów. Przeprowadzanie aktualizacji w godzinach pracy nie jest możliwe, gdyż wymagałoby to wstrzymania pracy pracowników, natomiast po godzinach pracy dlatego, że komputery są wyłączane (np. z powodu przepisów przeciwpożarowych).

7.2.2. Instalacja na serwerze plików

Drugim sposobem instalacji aplikacji jest umieszczenie jej plików na specjalnym komputerze pełniącym funkcję serwera plików. Aktualizacje są proste do przeprowadzenia, ponieważ sprowadzają się do przegrania plików w wyznaczone miejsce na tym serwerze.

Poważna wada ujawnia się natomiast podczas uruchamiania aplikacji. Proces ten wymaga wczytania do pamięci każdego komputera niespełna tysiąca plików o łącznym rozmiarze około dwudziestu megabajtów. Policzmy zatem jak długo operatorzy muszą czekać na uruchomienie przy następujących optymistycznych założeniach:

- w sieci pracuje dwadzieścia komputerów, z czego dziesięć jest uruchamianych równocześnie,
- faktyczna przepustowość sieci jest bliska maksymalnej i wynosi 8 megabitów na sekundę,
- przesyłanie małych plików wiąże się jedynie z dwukrotnym spowolnieniem przesyłania.

Sumaryczna liczba bajtów koniecznych do przesłania przy uruchamianiu wynosi

$$20MB * 10 = 200MB,$$

a czas potrzebny do ich przesłania:

$$\frac{200MB}{\frac{8Mb/s}{2}} = 400s = 6min40s.$$

Tak długie uruchamianie aplikacji jest niedopuszczalne.

7.2.3. Automatyczne aktualizacje

Trzecie z kolei rozwiązanie powstaje z połączenia dwóch poprzednich. Aktualizacja aplikacji polega na umieszczeniu na serwerze plików łatki, czyli zbioru plików, które się zmieniły od ostatniej wersji. Aplikacja jest zainstalowana na lokalnych dyskach i sama się aktualizuje podczas uruchamiania, którego pierwszym krokiem jest przegranie oraz zainstalowanie nowych łatek.

Wadą takiego rozwiązania jest to, że takie łatki mogą być duże, a czas ich instalacji zbyt długi. Szczególnie źle sprawdza się wykorzystanie technologii *WebStart*, gdyż wymaga ona łączenia plików w archiwa. Wystarczy, że jeden plik ulegnie modyfikacji, a konieczne będzie przesłanie całej grupy.

7.3. Rozwiązanie

Proponowane przeze mnie rozwiązanie jest odmianą automatycznych aktualizacji wykorzystującą zaprojektowany protokół niezawodnego rozgłaszania oraz sposób, w jaki pliki są wczytywane przez maszynę wirtualną Javy.

Wczytywaniem klas z plików do pamięci zajmuje się ładowarka klas (*ClassLoader*). Klasa jest wczytywana do pamięci dopiero wtedy, gdy jest po raz pierwszy wykorzystywana. Dzięki temu, pomimo tego, że opisywana aplikacja składa się z ponad dziesięciu tysięcy klas, podczas typowej pracy wczytywanych jest nie więcej niż połowa z nich.

7.3.1. Aktualizacje

Pliki wchodzące w skład aplikacji przechowywane są na dyskach lokalnych komputerów. Ponadto na każdym komputerze zapisany jest numer wersji. Przy każdym uruchomieniu aplikacja łączy się z serwerem w celu sprawdzenia czy nie ma nowych aktualizacji. Serwer porównuje wersję aktualną oraz przekazaną przez klienta i jeśli się różnią, to przesyła nowy numer wersji wraz z listą plików, które się zdezaktualizowały. Następnie klient usuwa wszystkie pliki z listy oraz zapisuje nowy numer wersji. Na tym kończy się aktualizacja. Za resztę odpowiedzialna jest specjalna ładowarka klas.

7.3.2. Ładowarka klas

Ładowarka klas na początku działa standardowo, czyli próbuje wczytać plik o nazwie wyznaczonej przez nazwę klasy do pamięci maszyny wirtualnej. Jeśli się to nie uda, ponieważ pliku nie ma, to wysyła do serwera żądanie przesłania go oraz czeka na odpowiedź.

Serwer odbiera żądania od klientów, a następnie je realizuje. Pliki wysyła posługując się protokołem niezawodnego rozgłaszania – wykorzystuje jeden lub kilka kanałów. Każdy plik wysyłany jest osobnym strumieniem, na który zapisywani są tylko ci klienci, którzy zażądali przesłania go.

Klienci odbierają wysyłane strumienie i zapisują odpowiednie pliki na dysku. Jeśli plik jest tym, którego przesłania zażądał klient, to odpowiednia klasa jest ładowana do pamięci. Najciekawsze jest to, że klienci podsłuchują strumienie zamówione przez innych klientów. Dzięki temu wystarczy, że dana klasa zostanie przesłana tylko raz – na prośbę jednego klienta. Pozostali, jeśli zdołali podsłuchać, będą mogli od razu załadować ją z dysku do pamięci.

Niniejsze rozwiązanie jest w dwóch punktach lepsze od rozwiązania polegającego na przesyłaniu łat:

- Podczas uruchamiania aplikacji te same dane są przesyłane mniejszą liczbę razy.
- Przesyłane są jedynie te dane, które są w danej chwili potrzebne.

Niewątpliwą wadą jest konieczność zastosowania serwera aktualizacji.

Rozdział 8

Testy

Przestawione w tym rozdziale testy mają na celu weryfikację zachowania protokołu w praktyce na podstawie prototypowej implementacji. W szczególności mają odpowiedzieć na następujące pytania.

1. Jaki wpływ ma liczba klientów na wydajność?
2. Jaki wpływ ma długość przesyłanych strumieni na wydajność?
3. Jaki wpływ ma wydzielenie kanałów na wydajność?
4. Czy „podsłuchiwanie" ma sens?

8.1. Sprzęt

Podczas testów miałem do dyspozycji pięć komputerów działających w sieci lokalnej. Cztery miały zbliżone parametry techniczne:

- procesor Intel Celeron 1100 - 1300MHz 256KB cache L2,
- pamięć operacyjna 512MB SDR RAM,
- karta sieciowa działająca w trybie przysłania danych z prędkością 10Mbit/s.

Jeden z nich pełnił funkcję serwera.

Piąty komputer miał zupełnie inną konfigurację:

- dwa procesory Intel Xeon 2.40GHz HT 512KB cache L2 z włączoną obsługą wielowątkowości (ang. *hyperthreading*),
- pamięć operacyjna 4GB DDR RAM,
- karta sieciowa działająca w trybie przysłania danych z prędkością 100Mbit/s.

Wszystkie komputery zostały połączone ze sobą za pomocą przełączników (ang. *switch*) w taki sposób, że maksymalna prędkość transmisji pomiedzy dowolnymi dwoma wynosiła 10Mbit/s.

8.2. Oprogramowanie

Każdy komputer działał pod kontrolą jednego z niżej wymienionych systemów operacyjnych:

- Microsoft Windows 2000 Professional,
- Microsoft Windows XP Professional,
- Microsoft Windows Server 2003.

Jedynym krokiem przygotowawczym do testów była instalacja maszyny wirtualnej, w związku z czym na komputerach mogły działać różne procesy. Na przykład na komputerze z dwoma procesorami w chwili testów zalogowanych było około dziesięciu osób i pomimo tego, że żadna z nich wówczas nie pracowała uruchomionych było kilkanaście aplikacji.

Środowiskiem uruchomieniowym Javy na wszystkich komputerach była najnowsza publicznie dostępna wersja, czyli J2SE 5.0 RC. Do maszyny wirtualnej nie były przekazywane żadne parametry sterujące jej pracą – działała z ustawieniami domyślnymi. Jedynym wyjątkiem było włączenie asercji.

Klasy zostały skompilowane za pomocą kompilatora wbudowanego w środowisko programistyczne Eclipse 3.0 z zainstalowaną wtyczką Cheetah 0.6. Prototyp nie był w żaden sposób optymalizowany pod kątem wydajności ani pod kątem wykorzystania pamięci. Skompilowane klasy zawierały rozszerzone informacje przydatne do odpluskwiania aplikacji.

8.3. Projekt

W celu uzyskania wiarygodnych wyników każdy test składał się z pięciu prób. Do zmierzenia czasu wykonania każdej z nich została użyta funkcja *System.currentTimeMillis()*. Pomiędzy próbami następowała dziesięciosekundowa przerwa. Każdy test związany był z jednym uruchomieniem maszyny wirtualnej (była zatrzymywana dopiero po przeprowadzeniu wszystkich prób testu).

Próba polegała na przesłaniu dziesięciu milionów bajtów danych. Różnice dotyczyły liczby klientów, długości i liczby strumieni oraz liczby kanałów.

Bufor kanału zawsze miał rozmiar 200KB, natomiast maksymalny rozmiar pakietu wynosił 8000 bajtów. Testy z różnymi rozmiarami pakietów też zostały przeprowadzone, ale w zupełnie innych warunkach, w związku z czym zostaną przytoczone tylko ogólne obserwacje.

Na serwerze, poza kilkoma aplikacjami niezwiązanymi z tematem tej pracy, działała wyłącznie aplikacja serwerowa. Aplikacje klienckie uruchamiane były tylko na czterech komputerach. W związku z tym podczas testów większej liczby klientów na każdym komputerze musiało zostać uruchomionych kilka aplikacji.

8.4. Test skalowalności

Test skalowalności polegał na wysyłaniu jednego strumienia do różnej liczby klientów. Trasmisja była w pełni niezawodna, czyli wszyscy klienci musieli otrzymać komplet danych.

Wyniki prób każdego testu zostały uśrednione i przedstawione w tabeli 8.1. Odchylenie standardowe było mniejsze od 10%, przy czym największe różnice w czasie wykonania wystąpiły pomiędzy pierwszymi a pozostałymi próbami testów. Sprawność to stosunek czasu potrzebnego do dostarczenia danych wielu klientom do czasu potrzebnego do dostarczenia danych jednemu klientowi. Wydajność to sprawność pomnożona przez liczbę klientów.

Liczba klientów	Czas (ms)	Sprawność	Wydajność
1	9 603	100,00%	100,00%
2	9 993	96,09%	192,19%
4	9 835	97,65%	390,58%
8	10 153	94,58%	756,66%
12	10 134	94,76%	1137,08%
16	10 700	89,75%	1435,94%
24	11 590	82,85%	1988,47%

Tabela 8.1: Wyniki testu skalowalności

Liczba strumieni	Rozmiar strumienia	Czas (ms)	Sprawność
1	10 000 000	9 835	100,00%
10	1 000 000	9 612	102,31%
100	100 000	9 687	101,52%
1 000	10 000	9 831	100,04%
10 000	1 000	12 256	80,24%

Tabela 8.2: Wyniki testu strumieni

Na ich podstawie tych można wysnuć następujące wnioski.

1. Czas przesłania danych do jednego klienta jest zbliżony do optymalnego, czyli protokołu tego można używać w zastępstwie TCP/IP.

2. Różnica w wydajności między jednym a dwunastoma klientami wynosi zaledwie 5%, natomiast między jednym a dwudziestoma czteroma – 11%.

Powyższe wyniki świadczą o tym, że protokół spełnia stawiane mu wymagania skalowalności.

8.5. Test strumieni

Test strumieni ma na celu zbadanie wpływu ich wielkości na szybkość trasmisji. Próby zostały przeprowadzone na strumieniach o wielkości od tysiąca do dziesięciu milionów bajtów. Dane były skierowane do czterech klientów, a ich łączny rozmiar w każdej próbie wynosił dziesięć milionów bajtów. Sprawność została obliczona w odniesieniu do przesyłania najdłuższego strumienia. Wyniki znajdują się w tabeli 8.2.

Wnioski:

- Strumienie o rozmiarze dziesięciu tysięcy bajtów lub dłuższe są przesyłane bez utraty wydajności.

- Przy przesyłaniu małych strumieni (o rozmiarze tysiąca bajtów) spadek wydajności wynosi zaledwie 20% – zysk z zastosowania protokołu w przypadku czterech klientów jest co najmniej trzykrotny.

- Wymaganie zachowania dużej wydajności przy przesyłaniu strumieni różnej długości zostało spełnione.

Próba	Czas pierwszego kanału (ms)	Czas drugiego kanału (ms)
1	10 000	10 625
2	9 578	10 031
3	9 375	10 093
4	10 110	10 922
5	9 766	10 641
Średnio	9 766	10 462
Czas jednego kanału		10 134
Strata wydajności	-3,63%	3,24%

Tabela 8.3: Wyniki testu kanałów

Spadek wydajności przy przesyłaniu małych strumieni wynika z konieczności utworzenia większej liczby małych fragmentów. Większość z nich ma ustawiony znacznik $SYNC$ i zawiera listę klientów wraz z ich numerami kolejnymi strumienia. Wszystko to sprawia, że serwer wykonuje znacznie więcej operacji przy wysyłaniu. Wydaje się, że optymalizacja implementacji umożliwiłaby poprawienie wyniku.

8.6. Test podsłuchiwania

Test podsłuchiwania miał na celu sprawdzenie czy klienci są w stanie skutecznie podsłuchiwać dane. Serwer wysłał tysiąc strumieni o długości dziesięciu tysięcy bajtów. Dane były skierowane do jednego klienta, pozostałych piętnastu podsłuchiwało.

Wyniki:

- Podsłuchujący klienci nie mieli wpływu na czas dostarczenia danych do klienta, do którego były skierowane, ponieważ nie wysłali żadnego pakietu do serwera.
- Spośród tysiąca wysłanych strumieni podsłuchujący klienci otrzymali w całości średnio 95%.

8.7. Test kanałów

Test kanałów miał na celu sprawdzenie w jaki sposób na wydajność i pewność transmisji wpływa zastosowanie kanałów, czyli równoległe przesyłanie więcej niż jednego strumienia jednym połączeniem.

Do dwunastu klientów wysłane zostały równolegle dwoma kanałami dwa strumienie o rozmiarze pięciu milionów bajtów każdy. Czasy poszczególnych prób zostały przedstawione w tabeli 8.3. Błąd pomiaru zamknął się w przedziale $\pm 20ms$.

Widać, że zastosowanie drugiego kanału wiąże się z utratą wydajności rzędu trzech procent. Nie jest to duża różnica, a zatem można stwierdzić, iż zastosowanie kanałów ma sens.

8.8. Wpływ rozmiaru pakietu na prędkość transmisji

Wybór odpowiedniego rozmiaru pakietu nie jest oczywisty. Przesyłanie danych o większym rozmiarze zwykle wiąże się ze wzrostem wydajności. Takie też wnioski można było wyciągnąć z następującego testu.

- Maksymalny rozmiar pakietu został ustalony na 64 tysiące bajtów (ograniczenie wynikające z zastosowania UDP).
- Serwer i sześciu klientów zostało uruchomionych na jednym komputerze.
- Serwer wysyłał do klientów strumień o długości dziesięciu milionów bajtów.

Wynik okazał się być lepszy od identycznego, ale przy wykorzystaniu pakietów o rozmiarze maksymalnie 8000 bajtów.

Sytuacja jednak diametralnie się zmieniła po umieszczeniu serwera i klientów na różnych komputerach. Okazało się, że:

- błędy transmisji i retransmisje były dużo częstsze niż przy zastosowaniu mniejszych pakietów,
- zapytań o status, jak i odpowiedzi było dużo więcej,
- czas transmisji wydłużył się ponad dwukrotnie.

Zbyt częste pytania o status i odpowiedzi spowodowane były zbyt dużym stosunkiem rozmiaru pakietu do rozmiaru bufora kanału.

Błędy transmisji wynikały ze zbyt małej pojemności bufora gniazda. Tolerancja na opóźnienia w pobieraniu z niego danych spadła niemalże do zera, co spowodowało gubienie pakietów przy niemal każdym uruchomieniu odśmiecarki maszyny wirtualnej.

Rozdział 9

Podsumowanie

W niniejszej pracy przedstawiłem istniejące protokoły niezawodnego rozgłaszania, ich zalety oraz wady. Zdefiniowałem wymagania, jakie powinien taki protokół spełniać, aby można go było wykorzystać w sieciach lokalnych. Następnie zaproponowałem własny protokół oraz przedstawiłem jego prototypową implementację. Na końcu wskazałem przykład zastosowania oraz przeprowadziłem testy. Teraz będę chciał dowieść, że protokół spełnia postawione wymagania w stopniu wystarczającym oraz przedstawić możliwe dalsze kierunki prac.

Sieć

Protokół działa w sieci lokalnej i wykorzystuje jej największe zalety, czyli sprzętowe rozgłaszanie, wysoką niezawodność oraz małe opóźnienia przesyłania danych. Wykorzystuje też fakt, że sieć jest zaufana, dlatego zupełnie pomija kwestię bezpieczeństwa.

Skalowalność

Testy wykazały, że protokół zachowuje wysoką efektywność niezawodnego dostarczania danych do przynajmniej dwudziestu czterech klientów. W takich warunkach zysk z jego zastosowania jest niemal dwudziestokrotny. Jestem przekonany, że optymalizacja implementacji może doprowadzić do dalszego wzrostu wydajności.

Czy wszyscy zainteresowani otrzymali dane?

Sterownik protokołu serwera dysponuje informacją czy wysłane dane zostały przekazane wszystkim aplikacjom klienckim. W przedstawionym rozwiązaniu wiedza ta nie jest udostępniana aplikacji działającej na serwerze, co można uczynić rozszerzając API. Wydaje mi się, że może taki właśnie powinien być dalszy kierunek prac.

Pełna lub częściowa niezawodność oraz wyróżnianie klientów

Rozwiązanie dostarcza dwóch poziomów niezawodności. Pełna jest zagwarantowana przez mechanizm potwierdzania dostarczonych danych, natomiast częściowa jest możliwa dzięki „podsłuchiwaniu".

Dołożone zostały starania, aby pomimo pracy w trybie częściowej niezawodności klient miał gwarancję otrzymania danych we właściwej kolejności oraz wiedział czy są kompletne. Ponadto ułatwiona została zmiana poziomu niezawodności poprzez wprowadzenie punktów synchronizacji. Testy wykazały, że koszt zachowania pełnej niezawodności nie jest zbyt duży, natomiast częściowa niezawodność (na poziomie 90%) – niemal darmowa.

Ograniczenia czasowe dostarczenia danych

Dane są dostarczane tak szybko, jak tylko można. Opóźnienia w dostarczeniu danych mogą być spowodowane wolnym działaniem klientów zwlekających w ich przetwarzaniu. Protokół dopuszcza tworzenie kanałów w obrębie połączenia, dzięki czemu czas dostarczenia danych jednym kanałem nie zależy od opóźnień w dostarczaniu danych innym.

Nie został zaimplementowany mechanizm umożliwiający odłączanie źle lub w ogóle nie działających klientów. Jest to konieczne do wykorzystania protokołu w praktyce.

Wiele serwerów

Protokół nie wprowadza żadnych ograniczeń co do liczby jednocześnie działających serwerów w jednej sieci, pod warunkiem jednak, że są od siebie w pełni niezależne. Konieczne jest skonfigurowanie każdego z nich tak, aby wykorzystywał inne adresy/porty.

Bezpieczeństwo

Problem bezpieczeństwa nie został poruszony w ogóle. Bardzo łatwo jest zakłócić działanie protokołu wysyłając odpowiednio spreparowany pakiet. Odporność na ataki można zwiększyć podpisując każdy przesyłany pakiet, jednak nie widzę możliwości całkowitego zabezpieczenia się przed nimi – zwłaszcza przed atakami typu „odmowa usługi" (ang. *denial of service*). Treść przesyłanych danych można ukryć szyfrując strumienie przed ich wysłaniem.

Wymagania pamięciowe

Testy zostały przeprowadzone z wykorzystaniem bufora kanału o rozmiarze dwustu kilobajtów. Nie więcej niż kolejne dwieście kilobajtów zajmowały bufory gniazd, pakietów oraz struktury pomocnicze. Wystarczyło to w zupełności do przesyłania danych do kilkunastu klientów siecią o przepustowości 10Mbit/s w pełni ją wykorzystując.

Ograniczenia przesyłanych danych

Protokół umożliwia przesyłanie dowolnych danych. Testy wykazały, że spadek wydajności przy wysyłaniu strumieni o wielkości tysiąca bajtów nie jest zbyt duży. Ograniczeniem jest wydajność serwera oraz klienta, a nie sam protokół, który łączy małe strumienie i wysyła je w jednym pakiecie. Wyższą sprawność można uzyskać optymalizując implementację. Spadek wydajności jest jednak nieunikniony z powodu dodatkowych informacji, jakie są związane z każdym strumieniem (każdy strumień ma tyle numerów, ilu jest klientów).

Działanie w niesprzyjających warunkach

Konieczność potwierdzania danych oraz retransmisje zapewniają prawidłowe działanie protokołu w niesprzyjających warunkach spowodowanych przez przestoje związane z pracą odśmiecarki pamięci.

API

Interfejs programistyczny bazuje na standardowych klasach *InputStream* oraz *OutputStream*, w związku z czym wykorzystanie protokołu jest proste i intuicyjne. Pomocna jednak mogłaby się okazać możliwość otrzymywania od klientów potwierdzeń przetworzenia danych oraz informacji o awaryjnym odłączeniu klientów. Przydatna byłaby też implementacja metody *flush*.

Niezależność od systemu operacyjnego

Prototyp został zaimplementowany w Javie. Nie wykorzystuje metod natywnych ani metod i klas, o których wiadomo, że mogą się różnie zachowywać w różnych systemach operacyjnych. Dzięki temu można go uruchomić na dowolnym systemie operacyjnym, na którym działa Java.

Java

Jednym z powodów wyboru Javy była chęć sprawdzenia czy możliwa jest implementacja w tym środowisku oraz na ile taki wybór utrudnia lub ułatwia to zadanie.

Prototyp i testy dowodzą, że możliwa jest implementacja w Javie. Największym ograniczeniem okazał się być brak kanałów służących do rozgłaszania, skutkiem czego są dodatkowe kopiowania pamięci ograniczające wydajność. Trudności związane z przestojami (w przypadku Javy spowodowanymi głównie pracą odśmiecarki) należałoby pokonać również w przypadku innej implementacji, chyba, że byłaby ona częścią systemu operacyjnego.

Wnioski

Przedstawiony protokół spełnia stawiane mu wymagania – jest wydajny i prosty w wykorzystaniu oraz okazuje się przydatny w praktyce. Implementacja jest jedynie prototypem, która służyła dowiedzeniu prawdziwości powyższego stwierdzenia. Wydaje mi się, że warto kontynuować pracę w celu stworzenia implementacji pozbawionej braków i niedostatków prototypu.

Dodatek A

Słowniczek

Serwer — aplikacja wysyłająca (rozgłaszająca) dane.

Klient — aplikacja odbierająca dane.

Połączenie — Reprezentuje możliwość przesyłania danych między serwerem a klientem. Szczegóły w punkcie 5.6.1.

Kanał — wydzielona część połączenia. Dzięki wydzieleniu kanałów w obrębie połączenia możliwe jest równoległe przesyłanie niezależnych danych. Szczegóły w punkcie 5.6.3.

Strumień — uporządkowany zbiór danych przesyłany w obrębie kanału. W danym momencie może być przesyłany tylko jeden strumień w obrębie jednego kanału. Każdy strumień ma swój początek i może, ale nie musi, mieć swój koniec. Szczegóły w punkcie 5.6.4.

Fragment — część składowa strumienia wysyłana w jednym pakiecie. Szczegóły w punkcie 5.6.5.

Dodatek B

Zawartość płyty CD załączonej do pracy

src — katalog zawierający źródła prototypu.

bin — katalog zawierający skompilowany prototyp.

readme.txt — plik tekstowy zawierający cenne wskazówki dotyczące prototypu, sposobu jego kompilacji i uruchamiania.

pracamgr.pdf — elektroniczna wersja niniejszej pracy.

Bibliografia

[RFC2887] M. Handley, S. Floyd, B. Whetten, R. Kermode, L. Vicisano, M. Luby, *The Reliable Multicast Design Space for Bulk Data Transfer*, RFC 2887, August 2000.

[Lane] R. G. Lane, *A Comprehensive Study of Reliable Multicast Protocols over Ethernet-Connected Networks*, 2000.

[RFC3453] M. Luby, L. Vicisano, J. Gemmel, L. Rizzo, M. Handley, J. Crowcroft, *The Use of Forward Error Correction (FEC) in Reliable Multicast*, RFC 3453, December 2002.

[PGM] J. Gemmell, T. Montgomery, T. Speakman, N. Bhaskar, J. Crowcroft, *The PGM Reliable Multicast Protocol*, IEEE Network special issue on "Multicasting: An Enabling Technology", Vol. 17, No. 1, Jan/Feb 2003, pp. 16-22.

[RFC3048] B. Whetten, L. Vicisano, R. Kermode, M. Handley, S. Floyd, M. Luby, *Reliable Multicast Transport Building Blocks for One-to-Many Bulk-Data Transfer*, RFC 3048, January 2001.

[JSR166] *JSR 166: Concurrency Utilities* http://www.jcp.org/en/jsr/detail?id=166

[JSR203] *JSR 203: More New I/O APIs for the Java™Platform ("NIO.2")* http://www.jcp.org/en/jsr/detail?id=203

[TNC] pkwooster, *Java Technology Forum: Taming the NIO circus*, http://forum.java.sun.com/thread.jsp?forum=4&thread=459338

[RFC793] *Transmission Control Protocol*, RFC 793, September 1981.