**Uniwersytet Warszawski**
Wydział Matematyki, Informatyki i Mechaniki

**Piotr Findeisen**
Nr albumu: 219427

# Multiplexer, wydajne narzędzie do komunikacji w systemach rozproszonych

**Praca magisterska**
**na kierunku INFORMATYKA**

Praca wykonana pod kierunkiem
**dr Janiny Mincer-Daszkiewicz**
Instytut Informatyki

25 sierpnia 2009

## Oświadczenie kierującego pracą

Potwierdzam, że niniejsza praca została przygotowana pod moim kierunkiem i kwalifikuje się do przedstawienia jej w postępowaniu o nadanie tytułu zawodowego.

Data Podpis kierującego pracą

## Oświadczenie autora pracy

Świadom odpowiedzialności prawnej oświadczam, że niniejsza praca dyplomowa została napisana przeze mnie samodzielnie i nie zawiera treści uzyskanych w sposób niezgodny z obowiązującymi przepisami.

Oświadczam również, że przedstawiona praca nie była wcześniej przedmiotem procedur związanych z uzyskaniem tytułu zawodowego w wyższej uczelni.

Oświadczam ponadto, że niniejsza wersja pracy jest identyczna z załączoną wersją elektroniczną.

Data Podpis autora pracy

## Streszczenie

W niniejszej pracy omówiono zagadnienie komunikacji w systemach rozproszonych. Opisano istniejące rozwiązania oraz zanalizowano ich przydatność w konkretnym zastosowaniu, jakim jest środowisko portalu Azouk. Przedstawiono także Multiplexer, nowe narzędzie do komunikacji dla systemów rozproszonych, stworzone na potrzeby Azouka i najlepiej spełniające jego wymagania. Multiplexer cechuje się dużą szybkością i minimalnymi opóźnieniami przy przesyłaniu komunikatów, odpornością na duże obciążenie i wysoką skalowalnością. W pracy omówiono implementację Multiplexera oraz porównano go z istniejącymi narzędziami. Przeprowadzono i opisano szczegółowe testy porównawcze Multiplexera ze Spreadem, najlepszym spośród konkurencyjnych rozwiązań.



## Słowa kluczowe

system rozproszony, komunikacja, serwis, usługa sieciowa, diagnostyka, skalowalność, kolejka komunikatów, aplikacja WWW, Spread, rozgłaszanie grupowe

## Dziedzina pracy (kody wg programu Socrates-Erasmus)

11.3 Informatyka

## Klasyfikacja tematyczna

D. Software
D.4 OPERATING SYSTEMS
D.4.4 Communications Management


## Tytuł pracy w języku angielskim

Multiplexer, an efficient communication tool for distributed systems

# Spis treści

# Wprowadzenie

W systemie rozproszonym kluczową kwestią jest komunikacja pomiędzy współpracującymi procesami, bez niej bowiem system nie tworzyłby spójnej całości. Komunikację można rozważać na różnych poziomach, począwszy od poziomu sygnałów elektrycznych przesyłanych po kablach tworzących sieć, poprzez poziom datagramów IP wymienianych pomiędzy węzłami sieci, datagramów UDP przesyłanych pomiędzy aplikacjami w systemie, strumieni TCP łączących te aplikacje, aż do poziomu logicznych komunikatów przez nie wymienianych. O ile nie zajmujemy się urządzeniami fizycznymi, nie ma potrzeby rozważać komunikacji na poziomie sygnałów elektrycznych i można ją traktować wyłącznie jako narzędzie używane przez warstwę IP i protokoły ją wykorzystujące. Twórcy aplikacji działających w systemie rozproszonym mogą zdecydować się na niskopoziomową obsługę gniazd sieciowych, korzystając bezpośrednio z protokołów UDP lub TCP, albo użyć jednego z wielu narzędzi do komunikacji – począwszy od prostych narzędzi ułatwiających korzystanie z gniazd sieciowych, poprzez narzędzia do serializacji i deserializacji danych, aż do skomplikowanych „agentów" obsługujących jednocześnie połączenia pomiędzy wieloma komunikującymi się aplikacjami.

Istnieją co prawda narzędzia do komunikacji o wolnodostępnym kodzie źródłowym, które są zarówno proste, jak i wygodne w obsłudze, jednak często nie zapewniają one w dostatecznym stopniu takich właściwości jak odporność na awarie, wysoka dostępność, skalowalność czy łatwa diagnostyka. Rozwiązania bardziej skomplikowane – zarówno te o wolnodostępnym kodzie źródłowym, jak i komercyjne – zazwyczaj lepiej spełniają te wymagania, jednak kosztem szybkości działania. Dodatkowo, wiele rozwiązań jest tworzonych z myślą o homogenicznych środowiskach – na przykład Java RMI wyłącznie dla Javy czy Windows Communication Foundation wyłącznie dla platformy .NET – lub wspiera tylko niektóre języki programowania.

W ramach niniejszej pracy zaprojektowano i wykonano Multiplexer, narzędzie do komunikacji, które ma umożliwiać wydajną wymianę komunikatów pomiędzy aplikacjami działającymi w systemie rozproszonym z węzłami połączonymi siecią TCP/IP. Nowe rozwiązanie ma być szybkie w działaniu oraz odporne na awarie sieci i węzłów, ale zapewnienie odporności na awarie nie powinno odbywać się kosztem wydajności. Dodatkowym wymaganiem jest skalowalność, Multiplexer ma obsługiwać równocześnie połączenia z wieloma procesami wykonującymi się potencjalnie na różnych węzłach systemu rozproszonego bez nadmiernego wykorzystywania zasobów systemowych oraz ułatwiać diagnozowanie i optymalizację systemu poprzez zbieranie i agregowanie informacji z różnych węzłów. Nowe narzędzie nie może zakładać homogeniczności środowiska, ma być dostępne dla aplikacji napisanych w Pythonie lub C++ i rozszerzalne na inne języki programowania oraz niezależne od platformy sprzętowej czy architektury procesora.

Praca podzielona jest na sześć rozdziałów. W rozdziale 1 przedstawiono cele, jakie ma spełniać komunikacja w systemach rozproszonych oraz zestaw właściwości, jakimi powinny cechować się narzędzia wspierające rozproszoną komunikację. Rozdział 2 zawiera szczegółowy opis środowiska portalu Azouk, na potrzeby którego powstał Multiplexer, wraz z opisem

konkretnych wymagań stawianych platformie komunikacyjnej, która miałaby to środowisko obsługiwać. W rozdziale 3 omówiono wybrane istniejące rozwiązania oraz zanalizowano je pod kątem posiadania pożądanych cech z rozdziałów 1 i 2. W rozdziale 4 opisano Multiplexer jako konkurencyjne narzędzie do komunikacji w systemach rozproszonych. Rozdział 5 zawiera porównanie Multiplexera ze Spreadem, istniejącym narzędziem do komunikacji najlepiej spełniającym wymagania portalu Azouk. Podsumowania pracy dokonano w rozdziale 6. Opis zawartości płyty CD, stanowiącej integralną część pracy i zawierającej między innymi kod źródłowy Multiplexera, znajduje się w dodatku A.

# Rozdział 1

# Komunikacja w systemach rozproszonych

W niniejszym rozdziale omówię podstawowe cechy systemu rozproszonego. Zanalizuję pożądane właściwości narzędzi do komunikacji w systemie rozproszonym oraz główne modele komunikacji.

Coulouris et al. [9] definiują system rozproszony jako zbiór komponentów umieszczonych na różnych komputerach połączonych siecią, które komunikują się i koordynują swoje działanie wyłącznie poprzez wymianę komunikatów. Oczywiste jest, że bez komunikacji system rozproszony byłby tylko zbiorem niepowiązanych komputerów, nie funkcjonowałby jako całość i nie można by z niego w żaden sposób skorzystać.

Autorzy książki wskazują na następujące pożądane cechy systemów rozproszonych:

- heterogeniczność, różnorodność komponentów (platform sprzętowych, systemów operacyjnych, zastosowanych języków programowania),
- otwartość, która pozwala na dodawanie nowych i usuwanie istniejących komponentów,
- bezpieczeństwo,
- skalowalność, możliwość dodawania nowych zasobów w celu zwiększenia możliwości systemu,
- wykrywanie i obsługa błędów (m.in. błędów sieci, niespodziewanego wyłączania się komponentów),
- przezroczystość położenia komponentów.

Ponieważ jedyną formą komunikacji poszczególnych komponentów między sobą jest wymiana komunikatów, odpowiedzialnością za zapewnienie tych cech obarczone będzie narzędzie obsługujące komunikaty w systemie.

Komunikację można rozważać na różnych poziomach – począwszy od sygnałów elektrycznych wysyłanych po kablach aż do logicznych komunikatów wymienianych między aplikacjami z użyciem protokołu opartego na TCP – i z uwzględnieniem zastosowanych mediów – od kabli sieciowych po łączność bezprzewodową. W niniejszej pracy rozważać będziemy narzędzia do komunikacji dla środowisk najczęściej spotykanych zarówno w zastosowaniach akademickich, jak i biznesowych, a więc dla systemów rozproszonych zbudowanych z użyciem sieci TCP/IP, opartych na połączeniach kablowych (a zatem wydajnych). Interesować nas będą skalowalne i otwarte systemy rozproszone z heterogenicznymi węzłami, ale od narzędzia do komunikacji nie będziemy wymagać zapewniania bezpieczeństwa. W praktyce bezpieczeństwo najczęściej

zapewnia się w warstwie sprzętowej i warstwie IP, poprzez odpowiednie odizolowanie systemu rozproszonego od świata zewnętrznego. Ponieważ problem przekazywania komunikatów jest istotnie różny od problemu przesyłania danych strumieniowych, zajmować się będziemy wyłącznie przekazywaniem komunikatów, jako częściej spotykanym w praktyce.

## 1.1. Pożądane właściwości narzędzi do komunikacji

W tej sekcji opiszę cechy, jakimi powinny charakteryzować się narzędzia do komunikacji w systemach rozproszonych. Oczywiście w konkretnym przypadku niekoniecznie wszystkie te właściwości są jednakowo pożądane, a niektóre mogą być nawet zbędne.

### 1.1.1. Wysoka dostępność

Wysoka dostępność usługi oznacza, że przerwy w jej działaniu (ang. *downtime*) są minimalne. W przypadku serwerów telekomunikacyjnych (ang. *carrier grade*) oczekiwana dostępność to 99.999% czasu, co w praktyce oznacza dopuszczalne przerwy w działaniu rzędu 5.26 minuty w ciągu roku. Jeżeli usługa jest stosowana wyłącznie na wewnętrzny użytek jakiejś organizacji, jej wysoka dostępność jest pożądana, ale nie jest elementem wizerunku tej organizacji. W przypadku, gdy usługa wpływa na jakość funkcjonowania witryny internetowej, jej wysoka dostępność staje się bardzo ważna, ponieważ witryna stanowi wizytówkę organizacji. Każda przerwa w jej działaniu może zostać zauważona przez użytkowników, a w przypadku popularnych serwisów informacja o niej rozpowszechniona. Na przykład problemy z dostępem do strony internetowej Naszej Klasy wypracowały dla firmy renomę nieprofesjonalizmu.

Narzędzia do komunikacji coraz częściej są nieodłącznym komponentem stron internetowych. W serwisach takich jak `www.google.com` czy `www.amazon.com` wyświetlana strona jest efektem działania kilku lub kilkudziesięciu odrębnych usług. Jeżeli usługi te połączone są wspólnym mechanizmem komunikacji, to przerwy w jego działaniu oznaczają przerwy w działaniu całego serwisu.

### 1.1.2. Pewność dostarczenia

Pewność dostarczenia komunikatu do odbiorcy ułatwia korzystanie z narzędzia do komunikacji. Nadawcy zazwyczaj zależy na tym, by jego wiadomość dotarła do odbiorcy. Jeżeli biblioteka do komunikacji nie dostarczałaby mechanizmu powiadamiania o otrzymaniu i ponownego wysyłania niedostarczonych wiadomości, aplikacja z niej korzystająca musiałaby robić to sama.

Pewność dostarczenia i wynikająca z niej łatwość użycia skutkuje dużo szerszym stosowaniem protokołu TCP niż UDP. Z drugiej strony, w pewnych zastosowaniach koszt związany z zapewnianiem gwarancji dostarczenia jest zbyt duży. W przypadku transmisji głosu i obrazu w rozmowach VoIP, zgubienie niektórych porcji danych może być nawet niezauważalne dla użytkownika.

### 1.1.3. Duża przepustowość

Przepustowość systemu to w ogólności liczba operacji, jakie może wykonać system w jednostce czasu. Kiedy system rozproszony spojony jest jednym narzędziem do komunikacji, ważne jest, by nie stało się ono tzw. „wąskim gardłem”, a więc tym komponentem, którego działanie

ogranicza przepustowość całego systemu. Na przepustowość narzędzia do komunikacji składa się wiele czynników: przepustowość sieci, liczba i szybkość procesorów obsługujących komunikację w sieci, ilość czasu procesora potrzebna na obsługę pojedynczej wiadomości, opóźnienia sieci, rozmiar buforów i inne.

### 1.1.4. Małe opóźnienia

Opóźnieniem narzędzia do komunikacji nazwiemy fizyczny czas (ang. *wall clock time*) pomiędzy wysłaniem wiadomości przez nadawcę a odebraniem jej przez odbiorcę. W przypadku komunikacji asynchronicznej, wielkość opóźnień ma małe znaczenie. Inaczej jest w przypadku komunikacji synchronicznej, np. przy generowaniu strony internetowej wspólnie przez różne komponenty systemu. Tam opóźnienia generowane przez poszczególne usługi, które wykonują zadania sekwencyjnie, sumują się i wchodzą w skład czasu generowania całej strony. Natomiast czas generowania strony ma istotny wpływ na satysfakcję użytkownika – użytkownicy preferują te serwisy, które działają tak szybko, jak szybko oni klikają.

### 1.1.5. Model publikacji-subskrypcji

W momencie pisania kodu generującego nową wiadomość w systemie, programista nie zawsze wie o wszystkich *przyszłych* odbiorcach tej wiadomości. W modelu, w którym to nadawca określa zbiór adresatów, tworzenie nowych komponentów, które mają odbierać istniejące wiadomości, wymaga zmian w kodzie jednego lub wielu programów. Proces ten jest podatny na błędy, ponieważ trudno zapewnić, że wykonane zostały wszystkie konieczne zmiany. W wyniku błędu nowe komponenty mogą otrzymywać tylko niektóre wiadomości.

W modelu publikacji-subskrypcji (ang. *publish-subscribe*) dla każdego rodzaju wiadomości definiuje się, które komponenty powinny takie wiadomości otrzymywać. Programista, pisząc kod generujący nową wiadomość w systemie, odpowiedzialny jest wyłącznie za określenie typu wiadomości (operacja *publish*). Stworzenie nowych komponentów wymaga jedynie zarejestrowania, którymi wiadomościami dane komponenty się interesują (operacja *subscribe*). Definicje typów wiadomości oraz subskrypcji mogą być podane zarówno statycznie, np. w pliku konfiguracyjnym, jak i dynamicznie, w trakcie działania systemu.

### 1.1.6. Automatyczna gwarancja spójności

Niektóre narzędzia do komunikacji gwarantują czas i kolejność dostarczenia wiadomości, dzięki czemu wszyscy odbiorcy wiadomości otrzymują identyczne sekwencje informacji. Jest to szczególnie istotne w sytuacji, gdy od spójnej aktualizacji stanu poszczególnych komponentów zależy poprawne działanie systemu. Przykładem takiego narzędzia jest Spread [23]. Istnieją także narzędzia do komunikacji obsługujące transakcje, a nawet transakcje rozproszone połączone z transakcjami w bazie danych (Websphere [15]). Problem spójności danych można także rozwiązywać nie na poziomie wymiany komunikatów, ale w sposobie korzystania z nich, jak to jest zrobione na przykład w Chubbym [8].

### 1.1.7. Zbieranie informacji diagnostycznych

W systemie rozproszonym zbieranie informacji o poszczególnych komponentach jest bardzo ważnie – niezbędne przy diagnostyce błędów, identyfikacji „wąskich gardeł" i optymalizacji systemu. Jeżeli komponenty używają standardowych mechanizmów logowania informacji, to

zbierane dane diagnostyczne i statystyczne zachowywane są na dyskach różnych komputerów, nie stanowią całości i nie dają się jako całość analizować. Gdy system działa niepoprawnie, analizowanie każdej jego części osobno może nie doprowadzić do znalezienia źródła problemu. W dużych systemach często jedno zadanie (np. żądanie strony WWW) jest realizowane poprzez kilka kolejnych serwerów. Jeżeli dojdzie do błędu, jego diagnoza może wymagać przejrzenia logów każdego z nich. Potrzebne jest zatem zbieranie informacji w jednym miejscu, z możliwością przeszukiwania i filtrowania. Pożądaną cechą takiej agregacji jest wspólne oznaczanie logów związanych z wykonywaniem pojedynczego zadania.

### 1.1.8. Rozszerzalność protokołu

Protokoły, w których wiele cech jest ustalonych – np. długość komunikatu, struktura komunikatu czy pozycja poszczególnych informacji w bajtowym zapisie komunikatu – są prostsze w implementacji i często szybsze. Jednak w praktyce stosowanie takich protokołów zazwyczaj prowadzi do problemów, gdy zachodzi potrzeba ich rozszerzenia. Użycie nieelastycznego protokołu powoduje, że poszczególne komponenty systemu, używające implementacji różnych jego wersji, nie mogą ze sobą współpracować. W przypadku narzędzia do komunikacji obsługującego system rozproszony, kompatybilność wstecz i w przód jest bardzo ważna, ponieważ równoczesna aktualizacja całego systemu wymagałaby jego wyłączenia i często w ogóle nie jest możliwa. Wymóg kompatybilności dotyczy nie tylko metadanych używanych przez to narzędzie do trasowania wiadomości, ale także samych wiadomości przez nie przekazywanych. Pożądane jest zatem, by narzędzie do komunikacji wspomagało tworzenie wiadomości w rozszerzalnym, elastycznym formacie.

### 1.1.9. Przenośność

W systemach rozproszonych heterogeniczność, możliwość stosowania różnorodnych komponentów, jest bardzo ważną cechą, ponieważ pozwala używać odpowiednich narzędzi, technologii i platform w zależności od ról, jakie mają pełnić poszczególne komponenty systemu. Narzędzie do komunikacji w systemie rozproszonym musi umożliwiać wymianę wiadomości pomiędzy różnorodnymi węzłami sieci. Protokół i sposób komunikacji oraz nałożone ograniczenia czasowe i pamięciowe nie mogą uzależniać tego narzędzia od konkretnego języka programowania, technologii czy platformy.

## 1.2. Modele komunikacji

W poprzedniej sekcji opisałem pożądane właściwości narzędzi do komunikacji. Oczywiście w niektórych zastosowaniach pewne cechy mogą być ważniejsze od pozostałych, w innych priorytety mogą być zupełnie inne. Niezależnie od poszczególnych właściwości narzędzi do komunikacji, pożądana może być komunikacja synchroniczna lub asynchroniczna.

### 1.2.1. Komunikacja synchroniczna

Model komunikacji synchronicznej jest realizowany wówczas, gdy odbiorca czeka na wiadomość nie wykonując w tym czasie innych czynności i utrzymując połączenie z nadawcą. Zauważmy, że gdy oczekiwanie na dane delegowane jest do osobnego wątku, którego jedynym zadaniem jest odebranie danych i przekazanie ich do wątku głównego, który w tym czasie

wykonuje inne zadania, komunikacja synchroniczna przekształca się w komunikację asynchroniczną.

Przykładem modelu synchronicznego w systemie rozproszonym jest komunikacja w prostej architekturze klient-serwer. Polega ona na wysłaniu żądania od klienta do serwera, a następnie wysłaniu odpowiedzi od serwera do klienta. Po wysłaniu żądania klient oczekuje na odpowiedź. Architektura ta została zastosowana w sieci WWW, w której przeglądarki pełnią rolę klientów, równocześnie i niezależnie proszących serwery o dane.

### 1.2.2. Komunikacja asynchroniczna

Komunikacja asynchroniczna występuje wówczas, gdy odbiorca przed otrzymaniem wiadomości wykonuje inne czynności lub nie utrzymuje stałego połączenia z nadawcą. Istotą asynchronicznej komunikacji jest to, że warstwa logiki aplikacji, prosząc inny komponent systemu rozproszonego o dane, nie czeka na nie bezczynnie, ale wykonuje inne czynności. Zauważmy, że gdy aplikacja w pętli wstrzymuje swoje działanie do czasu odebrania danych, komunikacja asynchroniczna przekształca się w synchroniczną.

Przykładem asynchronicznego schematu komunikacji są procesy w systemie UNIX używające do wymiany danych systemowych kolejek komunikatów.

Z punktu widzenia aplikacji, komunikacja asynchroniczna może być obsługiwana na dwa sposoby. W pierwszym modelu aplikacja czeka na zdarzenia związane z operacjami asynchronicznymi i sekwencyjnie je obsługuje (ang. *reactor pattern*). Taki rodzaj obsługi zdarzeń asynchronicznych jest wspierany bezpośrednio przez jądra systemów operacyjnych. W drugim modelu aplikacja, rozpoczynając operację asynchroniczną, np. żądanie o dane, definiuje, co ma się stać po jej zakończeniu (ang. *proactor pattern*). W tym modelu występuje odwrócenie sterowania (ang. *inversion of control*), ponieważ to biblioteka jest odpowiedzialna za wywołanie kodu aplikacji, ale jest on wygodniejszy w użyciu niż model pierwszy, ponieważ logika aplikacji jest tworzona podobnie jak w komunikacji synchronicznej. Taki rodzaj obsługi zdarzeń asynchronicznych nie jest bezpośrednio wspierany przez systemy operacyjne, a więc musi istnieć warstwa tłumacząca „reakcyjną" obsługę zdarzeń na „proakcyjną".

### 1.2.3. Podsumowanie

W tym rozdziale omówiłem pożądane cechy narzędzi do komunikacji, wymagania im stawiane oraz funkcje, jakie mają pełnić. W następnym rozdziale zaprezentuję portal Azouk, na którego potrzeby powstał Multiplexer, wraz z jego specyficznymi wymaganiami dla stosowanego narzędzia do komunikacji.

# Rozdział 2

# Wymagania

W poprzednim rozdziale zaprezentowałem ogólnie pożądane właściwości narzędzi do komunikacji i różne programistyczne modele korzystania z nich. W tym rozdziale omówię specyfikę portalu Azouk [5], stanowiącego wraz z całym zapleczem usług system rozproszony, na którego potrzeby powstał Multiplexer. Prezentacja działania tego portalu jest niezbędna dla uzasadnienia decyzji projektowych podjętych w trakcie tworzenia Multiplexera.

## 2.1. Portal Azouk

Portal Azouk [5] jest tylko z pozoru prostą stroną służącą do publikowania, opisywania znacznikami i wyszukiwania dokumentów. Użytkownicy mają możliwość umieszczania swoich dokumentów na serwerze Azouka oraz zapisywania tam odnośników do ciekawych stron. Podobnie jak w innych serwisach umożliwiających zapisywanie stron [10, 25], użytkownik może opisywać strony swoimi znacznikami. W Azouku użytkownik może dodatkowo udostępniać swoje dokumenty, nie będące stronami internetowymi – publikacje, książki, prace naukowe, etc. Tym, co wyróżnia Azouka spośród innych serwisów tego typu, jest semantyczne znacznikowanie. Użytkownik opisujący stronę lub dokument może używać nie tylko dowolnych prywatnych znaczników, ale może także korzystać z bogatej bazy predefiniowanych znaczników z określoną semantyką (tzw. *taxonomy tags*). Do dokumentu możemy przypisywać znaczniki określające jego typ, tematykę, autorów, powiązane firmy, etc. – przykład edycji znaczników dokumentu obrazuje rys. 2.1. Semantyczne znaczniki połączone z automatycznym znacznikowaniem każdego nowego dokumentu pozwalają na zbudowanie bazy danych (osób, dokumentów, stron, firm), w której można precyzyjnie wyszukiwać.

### 2.1.1. Struktura Azouka

Portal Azouk to przede wszystkim aplikacja webowa, korzystająca z wydzielonych usług. Decyzję o takiej architekturze systemu podjęto z następujących przyczyn: ze względu na niską wydajność niektórych operacji bazodanowych (np. przeszukiwania), niektóre żądania HTTP, kierowane do aplikacji, nie powinny być obsługiwane przez bazę danych, ale przez moduł przechowujący potrzebne informacje w pamięci. Ponieważ typowa aplikacja webowa FastCGI uruchamia od kilku do kilkunastu procesów, nie jest dopuszczalne, by każdy z tych procesów niezależnie inicjował taki moduł i przechowywał osobną kopię danych w pamięci – takie rozwiązanie powodowałoby bardzo duże sumaryczne jej zużycie. Ponadto, by zapobiec ewentualnym wyciekom pamięci, procesy aplikacji webowej często mają z góry ograniczony


**Rysunek 2.1:** Edycja znaczników dokumentu

czas życia. Przechowywanie indeksów w pamięci podręcznej aplikacji wymagałoby częstego ich budowania, co niepotrzebnie obciążałoby bazę danych i zasoby obliczeniowe. Tak więc wskazana jest separacja aplikacji webowej od serwisów dostarczających usługi, z których korzysta.

### 2.1.2. Usługi

Podział Azouka na aplikację webową i zbiór wspierających ją usług został wprowadzony na długo przed powstaniem Multiplexera. Początkowo wszystkie usługi oprogramowane były w Pythonie, a więc system rozproszony, jaki tworzyły, był homogeniczny. Do jego obsługi powstało narzędzie do komunikacji, również napisane w Pythonie, które miało umożliwiać prostą konfigurację środowiska (usługi nie musiały deklarować adresu IP i numeru portu, pod którym będą dostępne) i równoważenie obciążenia. Narzędzie to nie dawało żadnej gwarancji dostarczenia komunikatów, nie było odporne na awarie i – co ważne – wykorzystywało protokół oparty na Pythonowym mechanizmie serializacji danych, a więc nieprzenośny na inne języki programowania.

Usługa wyszukiwania umożliwia znajdowanie w bazie danych informacji na temat osób, dokumentów, stron i firm według podanych przez użytkownika znaczników. Wymaga ona wyjątkowo intensywnych obliczeń i dużej ilości pamięci, więc wkrótce okazało się, że, ze względu na wydajność, nie może być zaimplementowana w Pythonie. Ponieważ stosowane narzędzie do komunikacji nie umożliwiało bezpośredniego podłączania programów nie napisanych w Pythonie, usługa wyszukiwania, przepisana do C++, musiała pozostać opakowana kodem Pythonowym, umożliwiającym komunikację z innymi elementami systemu.

Ze względu na semantykę przypisaną do znaczników, wyszukiwanie odbywa się względem wybranych *znaczników*, a nie – jak w tradycyjnych wyszukiwarkach – względem niejednoznacznej *tekstowej frazy*. Dla przykładu, gdy użytkownik wpisuje skrót „SDP", Azouk umożliwia wybranie między „Service Delivery Platform", „Service Discovery Protocol" i innymi możliwościami. Oczywiście, żeby interfejs był wygodny w użyciu, podpowiedzi do wpisywanej frazy muszą być wyświetlane na bieżąco. Tak więc, zanim dojdzie do wyszukiwania, aplika-

cja musi wielokrotnie, przy każdej nowej literze wpisanej przez użytkownika, generować listy podpowiadanych znaczników (rys. 2.2). Wyliczanie podpowiedzi nie jest tak kosztowne jak wyszukiwanie, ale na czas ich generowania nałożone jest ściślejsze ograniczenie. Aby korzystanie z podpowiedzi było wygodne, muszą być one przekazywane użytkownikowi szybko, w czasie nie dłuższym niż 100 ms. Ponieważ narzut na obsługę żądania HTTP, pomijając czas wyliczania podpowiedzi, jest rzędu 40 ms, to uwzględniając czas potrzebny na przesłanie żądania i odpowiedzi przez internet, podpowiedzi powinny być wyliczane i przekazywane do aplikacji webowej w czasie nie dłuższym niż 10 ms. W ocenie architektów Azouka, niemożliwe jest szybkie generowanie podpowiedzi w oparciu o dane czytane za każdym razem z bazy danych. Takie rozwiązanie nie byłoby ani wydajne, ani skalowalne. Do osiągnięcia pożądanej wydajności potrzebne jest przechowywanie w pamięci wszystkich informacji niezbędnych do wygenerowania podpowiedzi. Podobnie jak w przypadku wyszukiwania, dane podręczne przechowywane są w pamięci wyspecjalizowanego serwisu dostarczającego usługę generowania podpowiedzi.

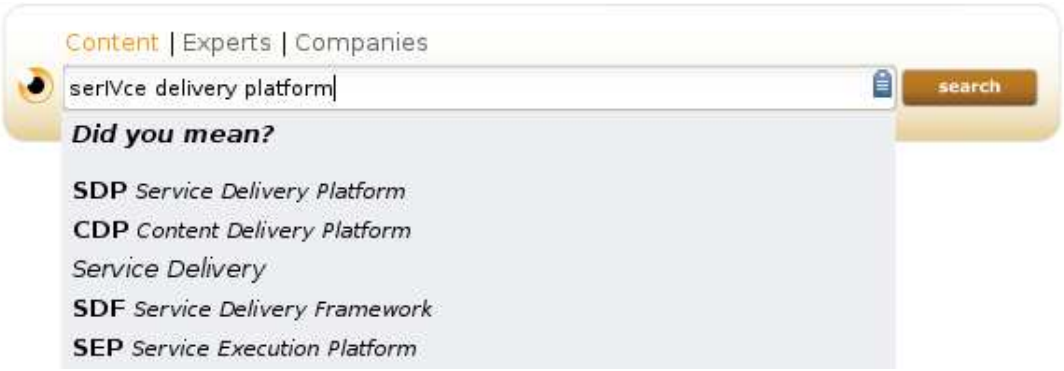

**Rysunek 2.2:** Podpowiedź do wpisywanego znacznika

Uzyskaliśmy więc trzy komponenty: aplikację webową, serwis wyszukiwania oraz serwis generowania podpowiedzi. Każdy z tych komponentów może występować w wielu instancjach, które trzeba połączyć tak, by każda instancja webowa była obsługiwana przez komplet serwisów pomocniczych oraz by serwisy sprawiedliwie dzieliły się pracą.

Wraz z rozwojem systemu okazało się, że usługi podpowiedzi i wyszukiwania nie są jedynymi komponentami, które należy wyodrębnić z aplikacji webowej. Gdy użytkownik wpisze frazę wyszukiwania, popełniając błąd w pisowni, niemożliwe jest wygenerowanie prawidłowych podpowiedzi. W takim przypadku niezależna usługa jest odpowiedzialna za wygenerowanie podpowiedzi najlepiej pasujących do wpisanej frazy, z uwzględnieniem potencjalnych błędów w pisowni (rys. 2.3).

Usługi podpowiedzi do wpisywanych znaczników i wyszukiwania to jedynie początek listy usług istniejących w systemie, które muszą się wzajemnie komunikować. Wraz ze wzrostem liczby komponentów, rośnie zapotrzebowanie na uniwersalne rozwiązanie komunikacyjne łączące je w jeden system rozproszony. W portalu Azouk następujące komponenty wyodrębnione są w postaci osobnych procesów:

**aplikacja webowa** napisana w Pythonie z użyciem biblioteki Django [11] aplikacja odpowiedzialna za interakcję z użytkownikiem poprzez przeglądarkę użytkownika; połączona z serwerem WWW poprzez FastCGI, z bazą danych oraz innymi usługami,




**Rysunek 2.3:** Podpowiedź do błędnie wpisanego znacznika

**baza danych** relacyjna baza danych, używana do przechowywania wszelkich danych trwałych; ze względu na wymagania istniejących bibliotek i narzędzi (np. Django), dostęp do tego komponentu musi być zawsze bezpośredni,

**Search** omawiana wcześniej usługa, odpowiedzialna za wyszukiwanie, tj. generowanie list dokumentów, osób, stron i firm pasujących do frazy podanej przez użytkownika, będącej zbiorem znaczników,

**Predict** omawiana wcześniej usługa, odpowiedzialna za generowanie podpowiedzi podczas wpisywania znaczników przez użytkownika,

**Oracle** omawiana wcześniej usługa, odpowiedzialna za rozpoznawanie znaczników we frazie błędnie wpisanej przez użytkownika,

**Email** usługa odpowiedzialna za wysyłanie poczty elektronicznej do użytkowników i administratorów portalu; odseparowana, by wysyłanie poczty nie blokowało działania aplikacji webowej,

**Log Collector** usługa odpowiedzialna za gromadzenie informacji diagnostycznych generowanych przez wszystkie komponenty systemu i umożliwiająca ich przeszukiwanie,

**Events Collector** usługa odpowiedzialna za spamiętywanie przez pewien czas zdarzeń rozgłaszanych w systemie i dostarczanie ich do komponentów systemu, które były nieaktywne w momencie rozgłaszania,

**Tagger** usługa odpowiedzialna za wyszukiwanie znaczników w dokumentach publikowanych w portalu (określanie autorów, tematyki, powiązanych firm) i określanie ich tytułu,

**Thumb NG** usługa odpowiedzialna za generowanie miniaturek dokumentów oraz podglądu pierwszych kilku stron (umożliwia wstępne zapoznanie się z dokumentem bez potrzeby ściągania całości); funkcjonalność generowania miniaturek obsługiwana jest przez dwa komponenty: zarządcę generowania (*Thumb Managera*), który przyjmuje od aplikacji webowej zlecenia dla nowych plików i wybiera z bazy danych istniejące pliki nie posiadające miniaturek lub podglądów, oraz serwery wykonujące faktyczne generowanie.

## 2.2. Podsumowanie wymagań

Wobec funkcji, jakie ma spełniać portal Azouk, konieczne jest zastosowanie architektury systemu rozproszonego, a więc i narzędzia do komunikacji umożliwiającego wymianę komunikatów między procesami znajdującymi się na różnych serwerach. Spośród ogólnie pożądanych

cech narzędzi do komunikacji, opisanych w sekcji 1.1, narzędzie dla portalu Azouk powinno mieć następujące właściwości:

- wysoka dostępność,
- pewność dostarczenia,
- małe opóźnienia,
- duża przepustowość,
- model publikacji-subskrypcji oraz
- łatwe zbieranie informacji diagnostycznych.

Ponieważ nie jest możliwe przewidzenie wszystkich zadań, jakie w przyszłości będzie pełnił system rozproszony portalu Azouk, zastosowany protokół komunikacyjny musi być rozszerzalny z zachowaniem kompatybilności ze starszymi wersjami.

W następnym rozdziale omówię istniejące narzędzia do komunikacji w systemie rozproszonym i przeanalizuję możliwość ich zastosowania w środowisku portalu Azouk.

# Rozdział 3

# Istniejące rozwiązania

W tym rozdziale omówię wybrane istniejące narzędzia do komunikacji. Dla każdego z nich, poza krótką charakterystyką, zamieszczę analizę przydatności w środowisku systemu rozproszonego Azouka. Celem tego rozdziału jest wybranie istniejącej implementacji narzędzia do komunikacji najlepiej dającej się zastosować w Azouku.

## 3.1. Punkt do punktu

Przy konstrukcji systemów rozproszonych bardzo często używa się narzędzi i bibliotek umożliwiających komunikację punkt do punktu (ang. *point-to-point*). Takie rozwiązania są proste w użyciu i w konfiguracji, a użytkownik może wybierać z bogatego zestawu dostępnych implementacji. Wolnodostępne są XML-RPC [33], Java RMI [14], CORBA [21], Thrift [22], usługi sieciowe, COM [7] i wiele innych.

### 3.1.1. XML-RPC

XML-RPC (ang. *XML Remote Procedure Call*) jest prostym protokołem zdalnego wywoływania procedur, używającym protokołu HTTP jako warstwy transportu i języka XML jako formatu kodowania danych. Jest poprzednikiem SOAP [32, 28], ale w odróżnieniu od SOAP jest naprawdę prosty w użyciu. Nie wymaga żadnego języka opisu interfejsów, a więc świetnie pasuje do idei szybkiego prototypowania (ang. *rapid prototyping*), za to gorzej nadaje się do aplikacji korporacyjnych, gdzie bardzo ważne jest dokładne specyfikowanie interfejsów. Przykładowo instancja klasy `MultiplexerMessage` (por. sekcja 4.3.2) zapisana przy użyciu kodowania XML-RPC wyglądałaby następująco:

```
<struct>
    <member>
        <name>id</name>
        <value><int>...</int></value>
    </member>
    <member>
        <name>from</name>
        <value><int>...</int></value>
    </member>
    <member>
        <name>to</name>
        <value><int>...</int></value>
    </member>
```

```
    <member>
        <name>type</name>
        <value><int>...</int></value>
    </member>
    <member>
        <name>message</name>
        <value><base64>...</base64></value>
    </member>
    <member>
        <name>timestamp</name>
        <value><int>...</int></value>
    </member>
    <member>
        <name>references</name>
        <value><int>...</int></value>
    </member>
    <member>
        <name>workflow</name>
        <value><base64>...</base64></value>
    </member>
</struct>
```

Ponieważ XML-RPC używa do zapisu danych języka XML i to w tekstowej postaci, z jego użyciem wiąże się duży rozmiar komunikatów (w przytoczonym wcześniej przykładzie jest to 520 bajtów, nie licząc białych znaków i właściwych danych), powodujący zbędne obciążenie sieci. Minimalne ze względu na rozmiar w bajtach wywołanie XML-RPC zdalnej metody z jednym parametrem typu `string` wiąże się z narzutem około 210 bajtów:

```
POST / HTTP/1.0
Host: localhost
User-Agent: a
Content-Type: text/xml
Content-Length: 131

<?xml version="1.0"?>
<methodCall>
<methodName>...</methodName>
<params>
<param><value>...</value></param>
</params>
</methodCall>
```

Dodatkowo, zastosowanie języka XML powoduje narzut obliczeniowy potrzebny na jego parsowanie i interpretację. Użycie tekstowego kodowania danych oznacza także problem przy przekazywaniu danych binarnych. Co prawda specyfikacja XML-RPC [33] zezwala na to, by pomiędzy znacznikami `<string>...</string>` znajdowały się dowolne bajty oprócz `<` i `&` (zapisywanych jako `&lt;` i `&amp;`), a więc teoretycznie zezwala na zapis dowolnych danych binarnych. Niestety, implementacja XML-RPC dostępna w standardowej dystrybucji Pythona (`xmlrpclib`) zawsze interpretuje XML jako zapisany w kodowaniu UTF-8, a zatem bajt 0 jest nielegalny, a bajt $10111111_2$ (w UTF-8 możliwy tylko wewnątrz wielobajtowego znaku) jest konwertowany na sekwencję $11000010\ 10111111_2$. Ze względu na rozbieżność między specyfikacją XML-RPC a implementacjami, przesyłanie binarnych danych wymaga zastosowania

kodowania base64, a więc i dodatkowego kosztu obliczeniowego na serializację i deserializację danych.

### 3.1.2. Java RMI

*Java Remote Method Invocation* jest Javowym mechanizmem wywoływania metod zdalnych obiektów, tj. obiektów znajdujących się w innej maszynie wirtualnej Javy, potencjalnie na innym serwerze. Poszczególne procesy korzystające z Java RMI komunikują się ze sobą, zapisując wymieniane dane przy użyciu Javowego mechanizmu serializacji. Java RMI jest specyficzne dla Javy i nieprzenośne na inne języki programowania, a więc nie nadaje się do użycia w środowiskach heterogenicznych. Jest za to bardzo wygodne, gdy wszystkie komponenty systemu rozproszonego napisane są w Javie.

Istnieje rozszerzenie Java RMI, RMI-IIOP (ang. *Remote Method Invocation over Internet Inter-Orb Protocol*), które umożliwia komunikowanie się Javowych programów używających RMI z programami napisanymi w innych językach. W RMI-IIOP zamiast Javowej serializacji używany jest format zapisu z CORBA i stosowanie tej technologii sprowadza się do używania CORBA w programach nienapisanych w Javie.

### 3.1.3. CORBA

*Common Object Request Broker Architecture* jest przenośnym między różnymi językami programowania i platformami mechanizmem zdalnego wywołania procedur. Jest to skomplikowana i ciężka w użyciu technologia [22], dobra do zastosowań korporacyjnych i do budowania systemów rozproszonych zawierających stare komponenty (ang. *legacy software*), posiadająca nowsze i lżejsze zamienniki, takie jak Thrift (opisany dalej). Mechanizm CORBA jest coraz rzadziej używany w nowych projektach oraz jest krytykowany za skomplikowaną, miejscami niespójną i nieprecyzyjną specyfikację, brak standardowego portu dostępu do usług (problemy z zaporami sieciowymi), skomplikowane odwołania nawet do obiektów znajdujących się lokalnie, model rozwoju standardu nie gwarantujący poprawy jego jakości, a jedynie zabezpieczenie interesów poszczególnych firm, wchodzących w skład komitetu rozwojowego [30].

### 3.1.4. Thrift

Thrift [22] jest zestawem narzędzi ułatwiającym pisanie serwisów sieciowych i łączenie się z nimi. W skład Thrifta wchodzi biblioteka dostarczająca obsługę połączeń sieciowych i serializację danych oraz klasy szkieletowe do pisania usług. Moduł serializacji danych obsługuje następujące typy danych: logiczne (`bool`), liczbowe (`byte`, `i16`, `i32`, `i64`, `double`), napisy (`string`, `binary`) oraz listy, zbiory i słowniki dowolnych typów. Korzystając z Thrift Interface Description Language (IDL), użytkownik może w wygodny sposób definiować własne struktury danych, podobnie jak w C (por. przykład 3.1). Słowo kluczowe `optional` definiuje opcjonalne pola nowej struktury, a numery pól są niezbędne do opisanego dalej wersjonowania definicji.

Drugą ważną cechą Thrift IDL jest możliwość definiowania interfejsów usług (zbiorów metod, podobnie do interfejsów w Javie) oraz generowania klas szkieletowych dla ich implementacji i użycia. Wygenerowane klasy klienckie są wygodne w użyciu i umożliwiają wykonywanie operacji synchronicznych i asynchronicznych. Operacja może być asynchroniczna tylko wtedy, gdy nie ma żadnej wartości zwracanej. Aplikacja kliencka może mieć równocześnie wiele

```
struct Message {
    // packet ID
    1: required i64 id,

    // ID of the sender
    2: i64 from,

    // recipient
    3: optional i64 to,

    // packet type
    4: required i32 type,

    // the actual message we want to send
    5: optional list<byte> message,

    // when the packet was sent
    6: optional i32 timestamp,

    // packet ID to which we reply
    7: optional i64 references,
}
```

**Przykład 3.1:** Użycie Thrift IDL

otwartych połączeń sieciowych z Thriftowymi usługami, ale połączenia te są niezależne i tylko jedno z nich może wykonywać operacje wejścia-wyjścia.

Bardzo ważną zaletą Thrifta jest kompatybilność wstecz i w przód – możliwość modyfikowania używanych typów danych bez konieczności zsynchronizowanej aktualizacji wszystkich stron komunikacji. Jest to możliwe dzięki temu, że formaty kodowania danych są samoopisujące i możliwe jest ich parsowanie bez znajomości definicji typów. Cecha ta jest naturalna np. dla dokumentów w języku XML, w których każda wartość związana jest z tekstową etykietą, a aplikacja nieznająca danej etykiety może ją po prostu zignorować. W binarnym kodowaniu Thrifta, dzięki zapisywaniu typów danych i numerów pól wraz z samymi danymi, osiągnięto te same właściwości. Jednak dzięki zapisywaniu właśnie numerów pól, a nie ich nazw, uzyskano format dużo mniej objętościowy niż XML.

Pod względem elastyczności struktur danych i wydajności protokołu ich kodowania, Thrift jest najlepszym z dotychczas omówionych narzędzi. Dzięki temu, że generowany kod wykonuje wywołania RPC w sposób blokujący, są one tak wygodne w użyciu, jak wywołania zwykłych lokalnych funkcji. Jednakże ta cecha jest też wadą, ponieważ powoduje, że w jednowątkowym programie może być wykonywana tylko jedna operacja wejścia-wyjścia naraz. W przypadku rozsyłania powiadomień do wielu odbiorców pojawia się problem skalowalności. Dodatkowo Thrift, tak samo jak inne rozwiązania punkt do punktu, nie zapewnia odporności na awarie sieci lub komponentów systemu i nie oferuje modelu publikacji-subskrypcji.

## 3.2. Narzędzia rozproszone

Ze względu na swoją architekturę, narzędzia oferujące komunikację punkt do punktu nie zapewniają odporności na awarie ani modelu publikacji-subskrypcji. Tak więc, w celu zapewnienia tych właściwości, konieczna jest odpowiednia implementacja w warstwie aplikacji lub skorzystanie z rozproszonych narzędzi do komunikacji.

### 3.2.1. Ruter

Część funkcjonalności wymaganej przez system rozproszony portalu Azouk można uzyskać odpowiednio konfigurując rutery lub przełączniki w sieci LAN łączącej komponenty systemu rozproszonego. Problem wysyłania wiadomości do wielu adresatów można rozwiązać w warstwie IP, poprzez zastosowanie rozgłaszania grupowego (ang. *multicast*). Aby móc używać rozgłaszania grupowego, komunikujące się serwery muszą znajdować się w jednej podsieci spiętej hubem lub przełącznikiem rozumiejącym rozgłaszanie bądź w różnych podsieciach połączonych ruterami z włączonym przekazywaniem datagramów IP rozgłaszania. Rozgłaszanie grupowe w warstwie IP działa w modelu publikacji-subskrypcji (por. sekcja 1.1.5). Przyszły odbiorca wyraża swoje zainteresowanie konkretnym „kanałem", identyfikowanym przez parę adres IP z klasy D i adres nadawcy. Swoje zainteresowanie zgłasza ruterowi jego podsieci, rozumiejącemu protokół IGMP (ang *Internet Group Management Protocol*) lub MLD (ang. *Multicast Listener Discovery*) dla IPv6, a ten rozgłasza je dalej tak, by zapewnić, że datagramy wysyłane przez nadawcę będą docierały także do tego nowego odbiorcy. Przy odpowiednim skonfigurowaniu sieci, rozgłaszanie grupowe oferuje transmisję danych od źródła do wielu odbiorców optymalną ze względu na poziom wykorzystania łącza.

Ze względu na jego bezpołączeniową naturę, nie jest możliwe połączenie rozgłaszania grupowego z protokołem TCP. W praktyce najczęściej wykorzystuje się protokół UDP, wcale nie zapewniając retransmisji zgubionych danych lub implementując ją w warstwie aplikacji. Stale prowadzone są prace nad skalowalnymi protokołami zapewniającymi retransmisję utraconych danych, istnieją propozycje takie jak *Real-time Transport Protocol*, *Resource Reservation Protocol*, *Scalable Reliable Multicast* i inne [31, 13]. Podstawowym problemem rozgłaszania grupowego jest brak ruterów umiejących trasować datagramy IP rozgłaszania i mających taką opcję włączoną. Przykładowo, na `ping 224.0.0.1` (adres `224.0.0.1` jest adresem wszystkich węzłów implementujących rozgłaszanie grupowe w warstwie IP) w sieci wydziałowej nie dostaje się żadnej odpowiedzi.

Możliwe jest także osiągnięcie funkcjonalności rozgłaszania grupowego bez jawnego jego stosowania. Odpowiednio skonfigurowany ruter może nie tylko przekazywać wiadomości od nadawcy do odbiorcy, ale dodatkowo duplikować cały ruch do jednego lub więcej „ukrytych odbiorców". Zastosowanie takiego rozwiązania przy budowie systemu rozproszonego wymaga posiadania odpowiedniego rutera i umiejętnego nim zarządzania. Każda zmiana w warstwie aplikacji, np. zdarzenie publikacji lub zapisu w modelu publikacji-subskrypcji, wymaga rekonfiguracji rutera. W praktyce urządzenia umożliwiające takie konfiguracje są drogie i wymagają wykwalifikowanych administratorów, a wszelkie problemy z konfiguracją w warstwie sprzętowej są trudne do zdebugowania. Opisana funkcjonalność jest najczęściej wykorzystywana przy testowaniu nowych wersji systemu (np. portalu internetowego) poprzez obciążenie ich kopią rzeczywistego ruchu, natomiast rzadziej jako komponent składowy samego systemu.

### 3.2.2. Java Message Service

*Java Message Service* jest specyfikacją interfejsu programistycznego narzędzi do komunikacji dla Javy. Implementacje JMS nie muszą używać zgodnych protokołów sieciowych i w praktyce nie da się równocześnie używać różnych implementacji. JMS przewiduje wysyłanie komunikatów bezpośrednio do odbiorcy oraz w modelu publikacji-subskrypcji.

Aby używać Java Message Service do łączenia komponentów napisanych w Javie z komponentami napisanymi w innych językach programowania, trzeba zastosować takie nie-Javowe narzędzie do komunikacji, które dostarczałoby komponentom Javowym interfejs zgodny z JMS

– w kontekście heterogenicznego systemu rozproszonego nie ma sensu rozważać samych implementacji Java Message Service. Przykładem narzędzia, które dostarcza interfejs JMS oraz interfejsy dostępne z innych języków programowania, jest OpenAMQP, opisany w kolejnym punkcie.

### 3.2.3. AMQP

*Advanced Message Queuing Protocol* [1] jest przenośnym, binarnym protokołem dla aplikacji wymieniających wiadomości. Protokół obejmuje operacje związane z wymianą wiadomości, trasowaniem, przekazywaniem, kolejkowaniem oraz wspiera model publikacji-subskrypcji. Pierwsze wersje protokołu zostały opracowane przez holding finansowy JP Morgan Chase oraz iMatix Corporation [29]. Mimo że protokół AMQP nie wywodzi się ze środowiska wolnego oprogramowania, specyfikacja protokołu nie jest związana z żadnymi patentami osób trzecich (nie zawiera rozwiązań opatentowanych) i nie ma przeszkód w tworzeniu implementacji tego protokołu o wolnodostępnym kodzie źródłowym [1]. W istocie, istnieją niezależne implementacje protokołu AMQP takie jak: OpenAMQP, Apache Qpid, Red Hat Enterprise MRG, RabbitMQ. Sesje, czyli konwersacje pomiędzy aplikacjami prowadzone z użyciem protokołu AMQP, są odporne na awarie sieci i pośredników, ponieważ po zerwaniu i ponownym nawiązaniu połączenia, aplikacje uzgadniają, które komunikaty zostały dostarczone, a które się zagubiły. W ten sposób gwarantowane jest, że każdy komunikat zostanie dostarczony i to dokładnie raz. Dodatkowo, komunikaty można oznaczać jako trwałe (ang. *persistent*) – wówczas serwer AMQP zapisze je na dysk przed potwierdzeniem odebrania, co daje pewność dostarczenia nawet w sytuacji rozległej awarii aplikacji, np. w przypadku braku prądu.

Oryginalna implementacja protokołu AMQP, OpenAMQP [19], jest dostępna na licencji GPLv3. Zawiera interfejs programistyczny dla C oraz dla Javy, kompatybilny z Java Message Service. Aby uzyskać dostęp do serwera AMQP z programów napisanych w Pythonie, należy skorzystać z biblioteki txAMQP [27]. Biblioteka txAMQP jest sterowana zdarzeniami, napisana z wykorzystaniem Pythonowej biblioteki Twisted [26], dzięki której można oczekiwać od txAMQP minimalnej liczby wątków systemowych i asynchronicznych operacji wejścia-wyjścia, a więc dobrej wydajności przy wielu równolegle wykonywanych operacjach. Niestety, zupełny brak dokumentacji biblioteki txAMQP, niemal zupełny brak komentarzy w kodzie i niedziałające przykłady uniemożliwiły przetestowanie współdziałania serwera OpenAMQP z klientem napisanym w Pythonie. Alternatywna implementacja protokołu AMQP dla Pythona, py-amqplib [20], choć wciąż w fazie beta, jest dobrze udokumentowana, ale za to jednowątkowa i synchroniczna. Natomiast biblioteka dostarczana przez projekt Qpid [3] tworzy co najmniej jeden wątek systemowy dla każdego połączenia, co ogranicza liczbę równoczesnych połączeń i niepotrzebnie zwiększa zużycie zasobów systemowych. Ze względu na ich architekturę, obie te biblioteki nie spełniają wymagań stawianych narzędziu do komunikacji, które miałoby być zastosowane w środowisku portalu Azouk.

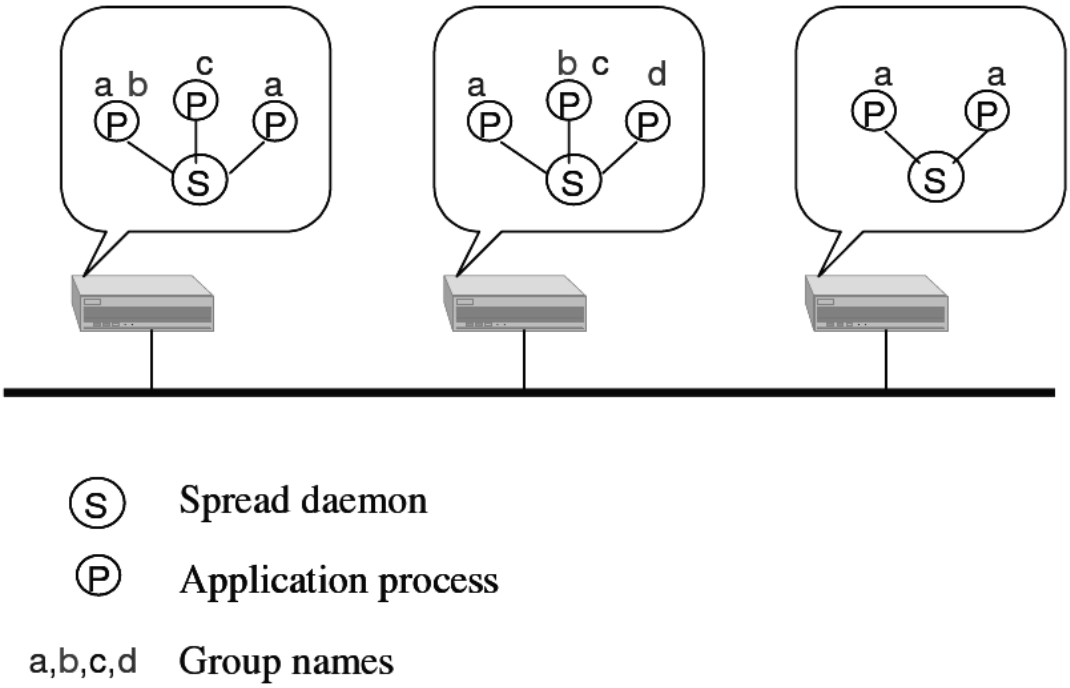
Zapewne OpenAMQP oraz inne implementacje serwerów AMQP dobrze działają w środowisku aplikacji napisanych w C/C++ lub Javie, jednak brak stabilnego wsparcia dla Pythona, realizującego asynchroniczne operacje wejścia-wyjścia, uniemożliwia zastosowanie AMQP jako narzędzia do komunikacji dla aplikacji napisanych w tym języku.

### 3.2.4. Spread

Spread jest zestawem narzędzi o otwartym kodzie, umożliwiających komunikację w systemach rozproszonych [23]. Według autorów projektu, Spreada cechuje wysoka wydajność i odpor-

ność na awarie sieci. Udostępnia on zarówno grupowe rozsyłanie wiadomości (ang. *multicast*), jak i przesyłanie wiadomości punkt do punktu. Użytkownik wysyłający wiadomość poprzez Spreada może – w zależności od ustawień danej wiadomości – nie otrzymać żadnych gwarancji dostarczenia, otrzymać gwarancję dostarczenia lub gwarancję dostarczenia z uwzględnieniem globalnie określonego porządku na wszystkich wiadomościach danej klasy (por. sekcja 1.1.6). Spread jest rozwijany przez firmę Spread Concepts, oferującą komercyjne wsparcie dla użytkowników, oraz wspierany przez grupy programistów z Johns Hopkins University i George Washington University. Spread był także finansowany przez amerykańskie agencje DARPA (Defense Advanced Research Projects Agency) oraz NSA (The National Security Agency).

Spread tworzy sieć o architekturze rozproszonej bez pojedynczego punktu porażki. Węzły sieci nie są ekwiwalentne i dzielą się na serwery (demony) i klientów. Serwery komunikują się między sobą, wymieniają wiadomości oraz, w razie konieczności, uzgadniają kolejność wiadomości. Każdy klient łączy się z jednym demonem Spreada pojedynczym połączeniem TCP. Spread nie zapewnia odporności na awarie połączenia pomiędzy klientem a serwerem, do którego ten klient jest podłączony, a jedynie na awarie połączeń między serwerami. Tak więc, żeby zapewnić pełną odporność na awarie sieci, konieczne jest instalowanie demona Spreada na każdym serwerze – wówczas klient łączy się tylko ze swoim lokalnym serwerem. Rysunek 3.2 przedstawia połączenia między poszczególnymi procesami w przypadku takiej właśnie konfiguracji. Demony Spreada łączą się w grupę roboczą i obsługują wspólną wirtualną przestrzeń kolejek komunikatów (grup), do których klienci mogą wysyłać wiadomości i z których mogą je odbierać. Dzięki takiemu schematowi komunikacji, Spread dostarcza model publikacji-subskrypcji (por. sekcja 1.1.5) oparty na dynamicznej konfiguracji.




rysunek pochodzi z dokumentu http://www.cnds.jhu.edu/pub/papers/cnds-2004-1.pdf

**Rysunek 3.2:** Architektura klient-demon Spreada

Demon Spreada napisany jest w języku C i działa pod systemami z rodziny UNIX i Windows (2000/NT/98/95/XP). Istnieją oficjalne implementacje biblioteki klienckiej dla C, Javy, Perla i Pythona oraz nieoficjalne, napisane przez użytkowników, dla C#, Ruby'ego, PHP i innych

języków programowania. Mimo niewątpliwie wielu zalet tego narzędzia, ostatnie wydanie Spreada miało miejsce w grudniu 2006, a interfejs dla Pythona – głównego języka programowania portalu Azouk – jest niezmieniony od lipca 2002. Brak nowych wydań może świadczyć o wstrzymaniu rozwoju projektu, ale może też – o stabilności implementacji i kompletności interfejsów.

Ze względu na architekturę bez pojedynczego punktu porażki oraz lekką i wydajną implementację, Spread jest niewątpliwie najlepszym kandydatem na narzędzie do komunikacji dla portalu Azouk. Zaletą Spreada jest wydajna biblioteka kliencka dla Pythona, zaimplementowana jako rozszerzenie Pythona w języku C. Ze względu na silne gwarancje dostarczenia wiadomości, Spread nie jest optymalnie dopasowany do środowiska aplikacji webowych, gdzie bardzo ważna jest minimalizacja opóźnień związanych z komunikacją – wiadomość, która dociera po 20 sekundach, zazwyczaj jest bezużyteczna lub nawet szkodliwa.

W następnym rozdziale opiszę Multiplexer jako narzędzie do komunikacji lepiej dopasowane do środowiska aplikacji webowych. Porównanie Multiplexera ze Spreadem znajduje się w rozdziale 5.

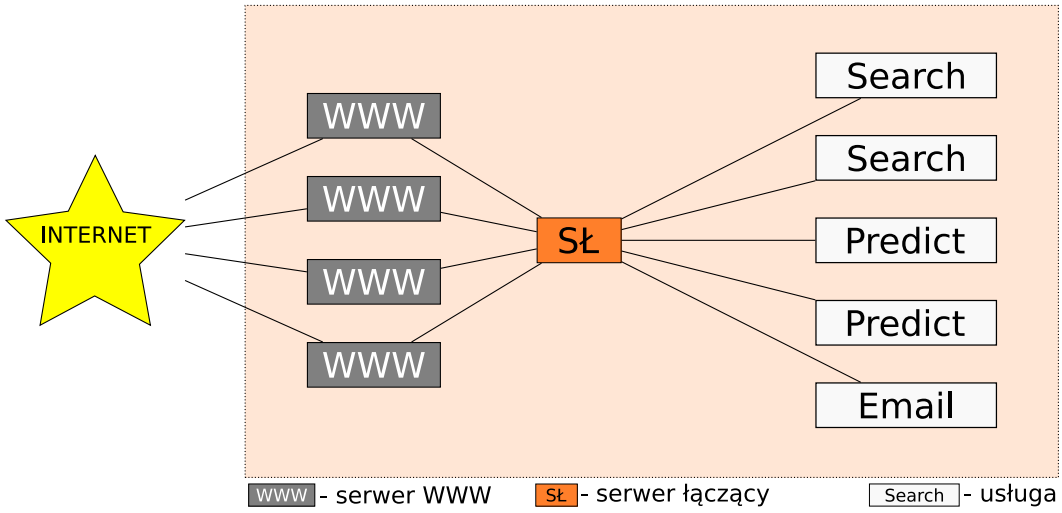
# Rozdział 4

# Multiplexer

W tym rozdziale przedstawię Multiplexer, nowe narzędzie do komunikacji dla systemów rozproszonych. Multiplexer powstał na potrzeby portalu Azouk, z powodu braku dostępnych implementacji, które byłyby zarówno szybkie, przenośne, jak i odporne na awarie.

## 4.1. Architektura

Jak już wspomniałem w rozdziale 2, portal Azouk obsługiwany jest przez aplikację webową i zestaw ośmiu usług, odpowiedzialnych za tworzenie stron HTML, generowanie w tle podglądów dokumentów itp. Skalowalność można zapewnić poprzez zwielokrotnianie instancji poszczególnych usług. Pociąga to jednak za sobą zwiększoną komplikację konfiguracji oraz problem równoważenia obciążenia. Problem konfiguracji jest tym większy, im więcej istnieje niezależnych instalacji systemu, bowiem dla każdej z nich trzeba utworzyć i utrzymywać osobny plik konfiguracyjny, którego aktualizacje muszą być zsynchronizowane ze zmianami w kodzie. Dla przykładu, w chwili obecnej jest 6 instalacji systemu Azouk, z czego 3 obejmują więcej niż jeden komputer, a 1 instalacja wykorzystuje aż 9 maszyn.

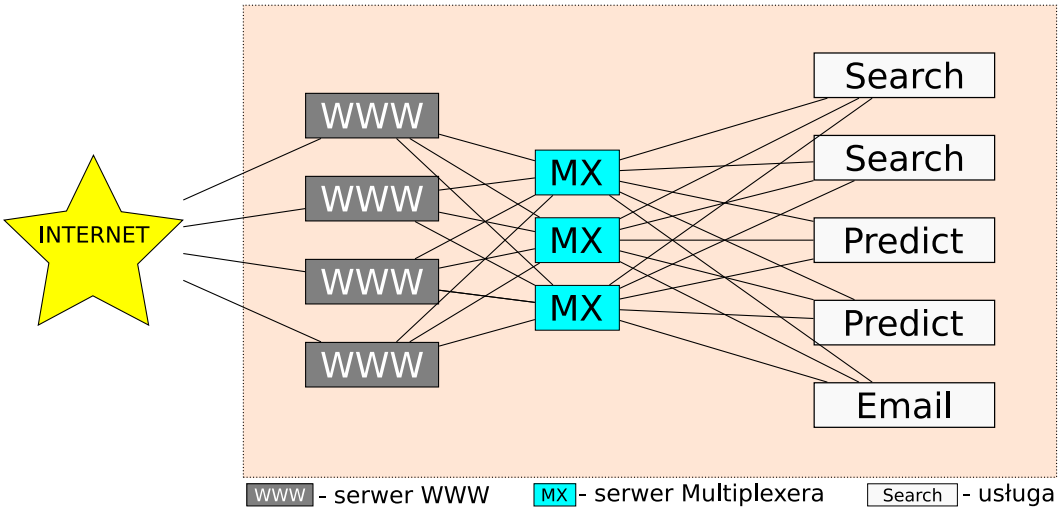



**Rysunek 4.1:** Konfiguracja „gwiaździsta” systemu

W celu uproszczenia konfiguracji przy jednoczesnym zapewnieniu równoważenia obciążenia, można zastosować model „gwiaździsty”, jak na rys. 4.1. W takiej architekturze serwer łą-

czący wszystkie komponenty, oznaczony SŁ, odpowiedzialny jest za obsługę żądań od aplikacji webowej i innych serwisów poprzez przekazywanie ich do właściwych usług. W pliku konfiguracyjnym dla instalacji wystarczy wpisać jeden adres – adres sieciowy, na którym nasłuchuje serwer łączący. Na ten adres zgłaszają się poszczególne usługi rejestrując swój typ i na ten sam adres aplikacja wysyła żądania, oznaczając je typem usługi, która powinna je zrealizować.

Taki model ma jednak poważną wadę. Jest to typowa architektura z centralnym serwerem, którego awaria powoduje przerwę w działaniu całego systemu (ang. *single point of failure*). Aby temu zapobiec, serwer centralny musi być zwielokrotniony, a każdy komponent musi się łączyć z co najmniej dwiema lub trzema jego instancjami. Z myślą o takiej właśnie konfiguracji został stworzony Multiplexer (rys. 4.2).




**Rysunek 4.2:** Konfiguracja bez pojedynczego punktu porażki

Architektura systemu bez centralnego serwera wymaga bardziej rozbudowanej biblioteki klienckiej, która radziłaby sobie z wieloma połączeniami naraz, z ponownym łączeniem się po zerwanym połączeniu oraz zapewniała odporność na awarie.

Zanim przejdę do omówienia protokołu komunikacyjnego i funkcjonalności biblioteki klienckiej, omówię technologie wykorzystane zarówno w serwerze Multiplexera, jak i w bibliotece, dzięki którym udało się zapewnić pożądane właściwości.

## 4.2. Wykorzystane technologie

### 4.2.1. Asio

W tradycyjnym modelu, w aplikacjach obsługujących wiele równoległych połączeń, każde gniazdo TCP obsługiwane jest w sposób synchroniczny przez osobny wątek. Dodatkowo istnieje wątek główny i jeden lub więcej wątków pomocniczych, odpowiedzialnych na przykład za regularne wykonywanie określonych czynności. Duża liczba wątków może prowadzić do wyczerpania zasobów systemowych i jest jedną z podstawowych przeszkód dla skalowalności. Ma to szczególne znaczenie w budowie serwera, który musi być w stanie obsługiwać nawet tysiące połączeń naraz, ale liczba wątków nie jest bez znaczenia także w bibliotece klienckiej. Jeżeli ograniczymy liczbę wątków do jednego, możemy zrezygnować z synchronizacji, w rezultacie zmniejszając czas dostępu do globalnych struktur danych.

```
// type of the buffer used
typedef std::vector<char> Buffer;
// callback to be called after read completes
void callback(boost::shared_ptr<Buffer> /*ignored*/,
        const asio::error_code& error, size_t bytes_transferred) {
    ....
}

// async_read invocation
boost::shared_ptr<Buffer> buffer(new Buffer(1024));
asio::async_read(socket, *buffer,
    boost::bind(callback, buffer, asio::placeholders::error,
    asio::placeholders:bytes_transferred));
```

**Przykład 4.3:** Przykład użycia Asio

Podstawą wydajnej pracy zarówno serwera Multiplexera, jak i biblioteki klienckiej, jest zastosowanie asynchronicznych operacji wejścia-wyjścia, obsługiwanych przez jeden tylko wątek. Asio (skrót od *asynchronous input-output*) [4] jest biblioteką dla C++ umożliwiającą odseparowanie asynchronicznych operacji od wątków je wykonujących. W szczególności możliwe jest zastosowanie Asio do obsługi wielu połączeń przy wykorzystaniu tylko jednego wątku. Asio wykorzystuje niskopoziomowe mechanizmy obsługi asynchronicznych operacji dostępne w systemie (w zależności od systemu: `epoll`, `kqueue`, `/dev/poll`, `select` lub `win IOCP`) i dostarcza wysokopoziomową warstwę abstrakcji, umożliwiającą asynchroniczne wykonywanie operacji wejścia-wyjścia, czekanie na upływ czasu czy odpytywanie serwerów DNS.

### Operacje asynchroniczne

Każda asynchroniczna operacja jako ostatni parametr przyjmuje wywołanie zwrotne (ang. *callback*), które będzie wykonane po zakończeniu operacji z parametrami określającymi sukces lub porażkę. Dla przykładu operacja odczytu danych z gniazda `socket` do bufora `buffer` wygląda następująco:

```
asio::async_read(socket, buffer, callback);
```

Ponieważ w C++ nie ma automatycznego odśmiecania, wykonując `async_read` musimy zagwarantować, że bufor na dane nie zostanie zniszczony przez zakończeniem odczytu, a więc przed wykonaniem wywołania zwrotnego. W przeciwnym przypadku może dojść do błędu pamięci z powodu zapisu do już zwolnionej pamięci bufora i wstrzymania całego programu. Aby zapewnić odpowiedni czas życia bufora, możemy trzymać go na zmiennej globalnej lub zawrzeć go w wywołaniu zwrotnym. W przykładzie 4.3 wywołanie zwrotne jest utworzone przy pomocy `boost::bind`, odpowiednika częściowej aplikacji znanej z języków funkcyjnych, i zawiera inteligentny wskaźnik na bufor, do którego wczytywane są dane. Trzymanie bufora jako zmiennej instancyjnej obiektu wykonującego odczyt jest niewystarczające, ponieważ sam ten obiekt może zostać zniszczony przed zakończeniem asynchronicznej operacji zwracanej.

### Odwrócenie sterowania

W bibliotece Asio nie ma globalnego stanu, a użytkownik może podzielić aplikację na fragmenty, z których każdy może w sposób niezależny używać tej biblioteki. Aby móc korzystać z Asio, niezbędne jest stworzenie obiektu klasy `asio::io_service`, który stanowi zarządcę asynchronicznych operacji. Dla przykładu, gniazdo TCP obsługiwane jest przez klasę

`asio::ip::tcp::socket`, której konstruktor przyjmuje referencję do obiektu `io_service`. Tak więc gniazdo sieciowe jest nieodłącznie związane z zarządcą asynchronicznych operacji, a użytkownik Asio musi zagwarantować, że czas życia obiektu `io_service` będzie dłuższy niż czas życia wszystkich gniazd sieciowych z nim związanych. `io_service` dostarcza metodę `run`, która w pętli wykonuje zlecone zadania i jest wywoływana przez wątki robocze.

Biblioteka Asio jest zaprojektowana zgodnie ze wzorcem projektowym odwrócenia sterowania (ang. *Inversion of Control*). Operacje asynchroniczne wykonywane są przez jeden lub więcej wątków roboczych działających w pętli. Po zakończeniu każdej operacji wywołanie zwrotne z nią związane jest zapisywane jako gotowe do wykonania. Wątki robocze, oprócz wykonywania operacji wejścia-wyjścia, są także odpowiedzialne za wykonywanie gotowych wywołań zwrotnych. W związku z powyższym, wyjątki podnoszone w wywołaniach zwrotnych nie propagują się do kodu otaczającego ich definicję, a raczej do kodu wywołującego metodę `run` obiektu `io_service`.

### 4.2.2. Google Protocol Buffers

Google Protocol Buffers [12] jest zestawem bibliotek i narzędzi służących do serializacji danych przenośnej pomiędzy językami programowania i architekturami sprzętowymi. Protocol Buffers powstał jako wewnętrzny projekt Google w 2001 roku, a jego kod został udostępniony w 2008 roku po kompletnym przepisaniu i usunięciu zależności od innych bibliotek rozwijanych przez Google. Protocol Buffers zawiera generator kodu i biblioteki czasu wykonania dla C++, Javy i Pythona. Istnieją także niezależne implementacje dla takich języków jak C, C#, Erlang, Haskell, Perl i innych.

Przenośna serializacja danych jest dobrą podstawą do budowy protokołów sieciowych i zdalnych wywołań procedur, jednak nie wszystkie niezbędne do tego komponenty zostały opublikowane wraz z Protocol Buffers. Tę właśnie lukę wypełnia Multiplexer, mimo że nie jest on stricte implementacją RPC.

#### Wiadomości

Typy danych obsługiwane przez Protocol Buffers są definiowane przy użyciu specjalnego języka opisu danych i interfejsów, podobnego do Thrift Interface Description Language (por. sekcja 3.1.4). Przykład 4.4 zawiera definicję klasy `Message` (por. przykład 3.1 dla Thrift IDL) w języku opisu interfejsów Google Protocol Buffers. Zauważmy, że w odróżnieniu od Thrift IDL, Protocol Buffers umożliwia deklarowanie pól o typach całkowitoliczbowych zarówno ze znakiem, jak i bez znaku.

Z definicji wiadomości `Message` z przykładu 4.4 program `protoc`, kompilator języka opisu danych Protocol Buffers, może wygenerować klasy `Message` w Javie, C++ i Pythonie o interfejsie bardzo podobnym, ale zgodnym z konwencją danego języka. Przykłady 4.5 i 4.6 obrazują podobieństwa i różnice w implementacjach Protocol Buffers dla różnych języków.

Implementacja Protocol Buffers dla Javy różni się tym od implementacji dla C++ i Pythona, że obiekty wygenerowanej klasy są niemodyfikowalne (por. przykład 4.7). Taka decyzja projektowa wynika z obserwacji, że w przypadku modyfikowalnych klas użytkownicy bardzo często tworzą lokalne ich kopie, nadmiernie obciążając zasoby systemu, by zagwarantować niezmienność danych i bezpieczeństwo współbieżnego odczytu przez różne wątki. W C++ wygenerowane klasy mogą być modyfikowalne, ponieważ zamierzony efekt można osiągnąć oznaczając nie modyfikowane instancje słowem **const**.

```
message Message {
    // packet ID
    required uint64 id = 1;

    // ID of the sender
    optional uint64 from = 2;

    // recipient
    optional uint64 to = 3;

    // packet type
    required uint32 type = 4;

    // the actual message we want to send
    optional bytes message = 5;

    // when the packet was sent
    optional uint32 timestamp = 6;

    // packet ID to which we reply
    optional uint64 references = 7;
}
```

**Przykład 4.4:** Język opisu Protocol Buffers

```
Message msg;
msg.set_id(random());
msg.set_type(SEARCH_REQUEST);
std::string serialized;
msg.SerializeToString(&serialized);
```

**Przykład 4.5:** Użycie Protocol Buffers w C++

### Rozszerzalność

Protokół serializacji obiektów Google Protocol Buffers został zaprojektowany z myślą o takich zastosowaniach jak protokół kodowania Thrifta (por. sekcja 3.1.4), a więc zapewnia kompatybilność, rozszerzalność wstecz i w przód. Protokoły binarne obu bibliotek zbudowane są w podobny sposób. Każdy obiekt zapisywany jest jako seria pól, a każde z pól zapisywane jest jako para: identyfikator (numer lub nazwa) i wartość. W obu bibliotekach wartości zapisywane są w ten sposób, że możliwe jest pominięcie przy dekodowaniu całego nieznanego pola, mimo że nie wiadomo, jak je deserializować.

Dzięki takim właściwościom protokołów binarnych, zarówno w Google Protocol Buffers, jak i w Thrifcie możliwe jest modyfikowanie używanych formatów danych bez konieczności równoczesnej aktualizacji wszystkich komponentów systemu rozproszonego, używających tych formatów.

Thrift obsługuje kilka protokołów serializacji (binarny, XML, JSON oraz tekstowy), a każdy z nich gwarantuje rozszerzalność. Twórcy Protocol Buffers przyjęli jednak inne założenia. Jak już powiedziałem, ich protokół binarny jest rozszerzalny, ale protokół tekstowy Protocol Buffers nie jest. Dzięki temu doskonale nadaje się do pisania plików konfiguracyjnych, gdzie nieznane parametry nie powinny być po cichu ignorowane.

Bardzo ważną zaletą Google Protocol Buffers jest sposób obsługi nieznanych pól napotykanych podczas deserializacji protokołu binarnego. Biblioteka Protocol Buffers dla Pythona

```
msg = Message()
msg.id = random()
msg.type = SEARCH_REQUEST
serialized = msg.SerializeToString()
```

**Przykład 4.6:** Użycie Protocol Buffers w Pythonie

```
Message msg = Message.newBuilder()
    .setId(random()).setType(SEARCH_REQUEST).build();
ByteString serialized = msg.toByteString();
```

**Przykład 4.7:** Użycie Protocol Buffers w Javie

działa podobnie jak Thrift, czyli ignoruje nieznane pola. Jednakże biblioteki dla C++ i Javy, bardziej rozwinięte od biblioteki Pythonowej, zachowują nieznane pola w osobnej strukturze danych. Dzięki temu aplikacja korzystająca z Protocol Buffers ma do nich dostęp i może je w jakiś sposób obsługiwać, chociażby zgłaszając administratorowi potrzebę aktualizacji. Ponadto podczas serializacji wiadomości, nieznane pola pochodzące z deserializacji również są zapisywane. Dzięki temu, w zastosowaniach takich jak serwer Multiplexera, a więc w aplikacjach, które czytają i przekazują wiadomości zakodowane przez Protocol Buffers, nie ma problemu „gubienia się" nieznanych pól, co jest niezbędne dla prawdziwej rozszerzalności.

#### Operacje wejścia-wyjścia

Implementacja Protocol Buffers dla C++ jest tworzona z myślą o wydajnych systemach obsługujących duże ilości danych, a więc niekopiowanie danych jest ważniejsze od wygody użytkownika. Operacje wejścia-wyjścia obsługiwane są przez podklasy klas `ZeroCopyOutputStream` i `ZeroCopyInputStream`. Interfejs klasy `ZeroCopyOutputStream` różni się od standardowych strumieni C++ brakiem operacji pisania małych porcji danych, które powodują wielokrotne kopiowanie obszarów pamięci. W zamian za to, `ZeroCopyOutputStream` dostarcza jedynie operację `Next(void **data, int* size)`. Po wywołaniu `Next *data` jest wskaźnikiem na bufor wielkości `*size`, który kiedyś zostanie zapisany. W celu minimalizacji liczby operacji kopiowania, wiadomości nie powinno się serializować poprzez metodę `SerializeToString`, jak w przykładzie 4.5, ale poprzez metodę `SerializeToZeroCopyStream`, która pisze kolejne bajty serializowanej wiadomości bezpośrednio do strumienia wyjściowego.

### 4.2.3. Boost

Boost [6] jest kolekcją wolnodostępnych i przenośnych bibliotek dla C++, pisanych przez ochotników z całego świata. Boost wypełnia braki w standardowej bibliotece C++, a niektóre biblioteki z Boosta mają wejść w skład standardu C++0x, następnej wersji języka C++. Mimo wysokiej jakości, Boost jest rzadko używany jako element publicznego API bibliotek. W skład Boosta wchodzi między innymi biblioteka inteligentnych wskaźników, kompensująca brak automatycznego odśmiecania w C++. Przy omawianiu Asio (por. przykład 4.3) pokazałem przykład użycia inteligentnego wskaźnika `boost::shared_ptr`, umożliwiającego współdzielenie obiektu i gwarantującego jego usunięcie w momencie skasowania ostatniej referencji do niego. Inteligentne wskaźniki wykorzystywane są powszechnie w implementacji Multiplexera i dzięki nim Multiplexer nigdy nie miał wycieków pamięci.

### Boost Preprocessor

Boost Preprocessor jest zestawem makr preprocesora C/C++ umożliwiających reprezentowanie struktur danych opisanych w języku preprocesora, generowanie kodu poprzez iterowanie po strukturach, zawierającym odpowiedniki pętli `for` i `while` etc. Boost Preprocessor jest używany przez Azouk Logging (por. sekcja 4.4.5) przy implementacji interfejsu dla C++ opartego na słowach kluczowych (ang. *keyword arguments*), a także przez bibliotekę obsługi sygnałów do definiowania niezbędnych struktur danych i funkcji dla każdego obsługiwanego sygnału.

Dla zaprezentowania możliwości Boost Preprocessor rozpatrzmy algorytm liniowego wyliczania n-tej liczby Fibonacciego zapisany w C w sposób następujący:

```
typedef struct linear_fib_state {
    int a0 /* i-th Fibonnaci number */, a1 /* i+1-ith Fibonnaci number */;
    int n; /* number of steps to go */
} linear_fib_state;

/* stop condition */
static int linear_fib_c(linear_fib_state p) { return p.n; }

/* calculating next step */
static linear_fib_state linear_fib_f(linear_fib_state p) {
   return (linear_fib_state){p.a1, p.a0 + p.a1, p.n - 1};
}

/* calculate n-th Fibonnaci number in O(n) */
int linear_fib(int n) {
   linear_fib_state p = {0, 1, n};
   while (linear_fib_c(p))
      p = linear_fib_f(p);
   return p.a0;
}
```

Przytoczony wcześniej algorytm można wyrazić w języku preprocesora C, korzystając z makr biblioteki Boost Preprocessor tak jak w następującym kodzie. Nie będę tu jednak analizować poszczególnych elementów tej konstrukcji, jest ona pokazana jedynie w celu zaprezentowania możliwości biblioteki.

```
/* linear_fib_state is just a 3-tuple (a0, a1, n) */
/* BOOST_PP_WHILE terminates when LINEAR_FIB_C expands to 0 */
#define LINEAR_FIB_C(d, p) BOOST_PP_TUPLE_ELEM(3, 2, p)

/* LINEAR_FIB_F calculates next step as (prev.a1, prev.a0 + prev.a1, prev.n - 1) */
#define LINEAR_FIB_F(d, p) (/* prev.a1 */ BOOST_PP_TUPLE_ELEM(3, 1, p), \
      /* prev.a0 + prev.a1 */ BOOST_PP_ADD_D(d, BOOST_PP_TUPLE_ELEM(3, 0, p), \
                                BOOST_PP_TUPLE_ELEM(3, 1, p)), \
      /* prev.n - 1 */ BOOST_PP_DEC(BOOST_PP_TUPLE_ELEM(3, 2, p)))

#define LINEAR_FIB(n) \
   BOOST_PP_TUPLE_ELEM(3, 0, BOOST_PP_WHILE(LINEAR_FIB_C, LINEAR_FIB_F, (0, 1, n)))
```

### Boost Python

Boost Python jest narzędziem do tworzenia rozszerzeń interpretera Pythona w C++. Oficjalny interpreter Pythona napisany jest w C, a więc API do definiowania jego rozszerzeń też jest w C. Trudność połączenia dynamicznego języka, jakim jest Python, z C powoduje, że

tworzenie rozszerzeń z bezpośrednim wykorzystaniem API interpretera nie jest łatwe i może prowadzić do błędów. Powstało wiele projektów, takich jak SWIG, SIP, Cxx, Weave czy Pyrex, których celem jest ułatwienie korzystania z Pythonowego C-API i zmniejszenie ryzyka błędów. Boost Python jest narzędziem łatwym w użyciu dla programistów C++ i nie wymaga żadnego dodatkowego kroku w procesie budowania: Boost Python składa się jedynie z linkowanej dynamicznie biblioteki oraz zestawu klas i funkcji korzystających z metaprogramowania szablonów C++. Boost Python dostarcza automatyczne przeciążanie funkcji, translację standardowych wyjątków, obsługę zwykłych i inteligentnych wskaźników. Boost Python nie obsługuje automatycznie klas z biblioteki STL (ang. *Standard Template Library*), ale dzięki metaprogramowaniu C++ można taką obsługę łatwo dodać.

Przykładowo, aby zdefiniować Pythonowy moduł `multiplexer` zawierający klasę `Server` o metodach `run` i `shutdown`, wystarczy utworzyć plik `multiplexer.cpp`, taki jak pokazano dalej i skompilować go do biblioteki dzielonej `multiplexer.so`.

```
#include <boost/python.hpp>
#include "server.h" // definition of class Server
BOOST_PYTHON_MODULE(multiplexer) {
    boost::python::class_<Server>("Server")
        .def("run", Server::run)
        .def("shutdown", Server::shutdown);
}
```

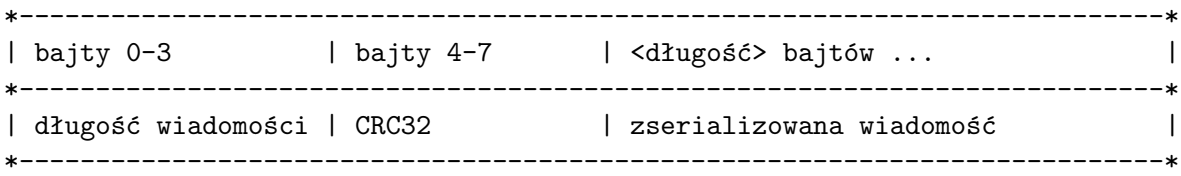
## 4.3. Protokół

Protokół komunikacyjny między serwerem Multiplexera a biblioteką kliencką stworzony jest w oparciu o Google Protocol Buffers. Jak napisałem w sekcji 4.2.2, Protocol Buffers nie dostarcza wsparcia dla *strumieni wiadomości*. W szczególności operacja parsowania wiadomości ze strumienia wczytuje wszystkie dostępne dane aż do osiągnięcia końca strumienia. Ze względu na kompatybilność wstecz biblioteki Protocol Buffers istnieje możliwość jej „oszukania" poprzez wstawianie specjalnego znacznika `END_GROUP` po każdej wiadomości. Oczywiście takie rozwiązanie nie jest oficjalnie wspierane i nie daje gwarancji na kompatybilność w przód. Dodatkowo biblioteka Protocol Buffers nie dostarcza żadnego wsparcia dla operacji asynchronicznych, niezbędnych dla skalowalności. Wobec takich ograniczeń konieczne jest zastosowanie buforowanego (asynchronicznego) odczytu pojedynczych wiadomości i późniejsza ich deserializacja.

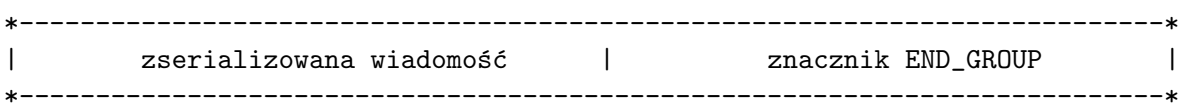
### 4.3.1. Format sieciowy

Na poziomie gniazd sieciowych pojedyncza wiadomość wymieniana w ramach protokołu Multiplexera wygląda jak przedstawiono na rys. 4.8. *Zserializowana wiadomość* oznacza wiadomość `MultiplexerMessage` (por. sekcja 4.3.2), zdefiniowaną i zserializowaną przy użyciu Google Protocol Buffers. Długość i suma kontrolna zserializowanej wiadomości zapisane są jako liczby bez znaku począwszy od najmniej znaczących bajtów (ang. *little endian*).

Protokół Multiplexera jest w całości zorientowany na wymianę wiadomości, poza nimi nie są przekazywane żadne inne dane. Z punktu widzenia aplikacji protokół oparty na formacie sieciowym przedstawiony na rys. 4.9 byłby równie użyteczny, ale utrudniałby buforowany odczyt, a także nie dostarczałby żadnej kontroli poprawności. W szczególności nie byłoby możliwe odróżnienie wiadomości pochodzących od biblioteki klienckiej od losowych danych wysyłanych przez przypadkowy proces.




**Rysunek 4.8:** Protokół sieciowy Multiplexera




**Rysunek 4.9:** Prostszy, alternatywny protokół

Zauważmy, że w tak zdefiniowanym protokole jest bardzo niewiele niezmiennych części, a więc jest on bardzo elastyczny. Dodatkowo, jego obsługa na poziomie sieciowych oktetów wymaga jedynie umiejętności liczenia sumy kontrolnej CRC32 i zapisywania danych całkowitoliczbowych bez znaku w porządku od najmniejszego bajtu. Do liczenia sum kontrolnych wybrano algorytm CRC32, ponieważ jest on powszechnie stosowany i istnieje wiele bibliotek dla różnych języków programowania dostarczających jego implementację.

Biblioteka Protocol Buffers zapewnia rozszerzalność protokołu poprzez modyfikowanie definicji klasy `MultiplexerMessage`.

### 4.3.2. MultiplexerMessage

Serwer Multiplexera komunikuje się z biblioteką kliencką wymieniając wiadomości klasy `MultiplexerMessage`, zserializowane przy użyciu Google Protocol Buffers i zapisane w formacie sieciowym opisanym w poprzedniej sekcji. Pełna definicja tej klasy znajduje się w dodatku B, tutaj omawiam jedynie najważniejsze jej pola.

**type** najważniejsze pole, opisuje typ wiadomości, dzięki któremu serwer Multiplexera wie, komu przekazać daną wiadomość (por. sekcja 4.3.3); również dzięki polu `type` odbiorca wie, co z zrobić z daną wiadomością,

**to** alternatywnie do definiowania typu wiadomości, można jawnie zdefiniować odbiorcę, przekazując w polu `to` jego identyfikator. Jeżeli nie wypełnia się pola `type`, istotne jest, by odbiorca w inny sposób dowiedział się, co zrobić z daną wiadomością. Odpowiada za to pole `references`,

**from** zawiera identyfikator nadawcy wiadomości i jest wykorzystywane do wypełniania pola `to` przy odpowiadaniu na daną wiadomość,

**id, references** unikatowy, losowy identyfikator wiadomości, dzięki któremu pojedyncza wiadomość jest dostarczana co najwyżej raz do pojedynczego odbiorcy; odbiorca, odpowiadając na wiadomość, wpisuje jej `id` w polu `references`,

**message** treść wiadomości, może to być dowolny ciąg bajtów; pole `message` nie jest w żaden sposób interpretowane przez serwer Multiplexera i jest przekazywane niezmienione do odbiorcy,

**workflow** identyfikator przepływu, z którym związana jest dana wiadomość (por. sekcja 4.4.5).

Wiadomości `MultiplexerMessage` służą nie tylko do komunikacji pomiędzy biblioteką kliencką a serwerem Multiplexera, ale także do komunikacji pomiędzy poszczególnymi komponentami

systemu. Zwykłe wiadomości, a więc takie, które serwer przekazuje dalej, nie są modyfikowane i w niezmienionej formie trafiają do odbiorcy. W szczególności pola wiadomości `MultiplexerMessage`, których serwer nie zna, są przekazywane niezmodyfikowane do odbiorcy. Co prawda taką funkcjonalność dostarcza biblioteka Protocol Buffers, jednak w serwerze Multiplexera została ona osiągnięta w prostszy sposób. Po odebraniu wiadomości, serwer deserializuje ją i zachowuje zarówno zdeserializowaną wiadomość, jak i jej zapis w sieciowym formacie. W ten sposób, przy przesyłaniu wiadomości do odbiorcy, serwer nie musi jej ponownie serializować, dzięki czemu zmniejsza się zużycie procesora.

Dzięki zastosowaniu Google Protocol Buffers do opisu metadanych wiadomości przekazywanych przez Multiplexer, wszystkie jego komponenty zapewniają kompatybilność w przód pod warunkiem przestrzegania prostych reguł. Jeśli klasie `MultiplexerMessage` nie dodamy nowych pól wymaganych i nie zmienimy istotne typów jej atrybutów (przykładowo, zmiany z `uint32` na `uint64` lub z pola `optional` na `repeated` są dopuszczalne), to niezaktualizowane komponenty będą mogły bez problemu współpracować ze zaktualizowanymi. Tym samym biblioteka Protocol Buffers gwarantuje rozszerzalność protokołu.

### 4.3.3. Trasowanie

Serwer Multiplexera podejmuje decyzję o trasowaniu wiadomości na podstawie kilku czynników. Niektóre wiadomości specjalne (np. `HEARTBIT`, `BACKEND_FOR_PACKET_SEARCH`, `CONNECTION_WELCOME`) są obsługiwane wewnętrznie przez serwer. Wiadomości z wypełnionym polem `to` przekazywane są bezpośrednio do odbiorcy. Dla wiadomości bez wypełnionego pola `to`, pole `type` służy do odnalezienia odpowiednich reguł trasowania. Reguły trasowania dla określonego typu wiadomości to lista rekordów, z których każdy określa, do jakiego typu odbiorcy należy przekazać daną wiadomość oraz czy do jednego, czy do wszystkich odbiorców danego typu. Pojedyncze rekordy trasowania określają także, czy nadawca ma być powiadamiany o niemożności przesłania wiadomości według danej reguły (np. o braku podłączonych odbiorców danego typu).

Reguły trasowania definiowane są statycznie w pliku konfiguracyjnym, wczytywanym przez serwer przy starcie. W przyszłości możliwe będzie przesyłanie reguł trasowania przez sieć, w celu zrekonfigurowania serwerów Multiplexera bez konieczności ich restartu. Format pliku konfiguracyjnego Multiplexera jest tekstowym formatem serializacji klasy `MultiplexerRules`, a zatem nie ma konieczności definiowania własnego parsera do jego czytania, jest on bowiem dostarczany przez bibliotekę Protocol Buffers. Ponieważ wczytany plik konfiguracyjny reprezentowany jest jako wiadomość Protocol Buffers, możliwe jest przesyłanie go przez sieć w nietekstowym, binarnym formacie.

W razie potrzeby możliwe jest zadeklarowanie osobnych reguł trasowania dla pojedynczej wiadomości. Służy do tego pole `override_rrules` klasy `MultiplexerMessage`, które jest listą rekordów trasowania w formacie identycznym do użytego w pliku konfiguracyjnym. Jeżeli to pole jest wypełnione, serwer Multiplexera nie korzysta z wbudowanych reguł trasowania, a zamiast nich używa reguł przekazanych wraz z wiadomością.

## 4.4. Właściwości

Spośród cech narzędzi do komunikacji wymienionych w sekcji 1.1, Multiplexer zapewnia te, które są ważne dla implementacji portalu Azouk. W szczególności Multiplexer nie dostarcza automatycznej gwarancji spójności (por. sekcja 1.1.6), a zamiast tego zapewnia małe

opóźnienia, dużą przepustowość i inne cechy szczególnie pożądane w środowisku portalu internetowego. Automatyczna gwarancja spójności może być zapewniana przez wyższe warstwy, korzystające z Multiplexera.

### 4.4.1. Wysoka dostępność

W sekcji 1.1.1 uzasadniłem, dlaczego wysoka dostępność narzędzi do komunikacji jest bardzo ważna. Od wysokiej dostępności warstwy komunikacji zależy wysoka dostępność całych serwisów i portali, a od tej z kolei reputacja firm je utrzymujących. Ma to szczególne znaczenie w przypadku firm takich jak Google, Amazon czy Yahoo, których model biznesowy to oferowanie wysokodostępnych i niezawodnych usług w internecie.

Azouk z założenia również ma być niezawodnym portalem, przeznaczonym dla profesjonalistów z branży telekomunikacyjnej. Użytkownikami Azouka będą ludzie pracujący na co dzień z rozwiązaniami telekomunikacyjnymi, cechującymi się wysoką dostępnością, a więc wszelkie przerwy w działaniu portalu będą miały bardzo negatywny wpływ na reputację.

Multiplexer zapewnia wysoką dostępność na dwóch poziomach. Po pierwsze, serwery Multiplexera mogą być dublowane, a nawet mogą występować w dowolnie wielu instancjach. Dzięki temu architektura systemu zbudowanego z wykorzystaniem Multiplexera nie ma pojedynczego punktu awarii oraz jest odporna na błędy sieci i przerwy w działaniu całych serwerów.

Po drugie, Multiplexer dostarcza mechanizmy *event* i *query*, które zapewniają funkcjonalność „rozproszonego RPC". Oba mechanizmy opisane są w kolejnej sekcji.

### 4.4.2. Pewność dostarczenia

Multiplexer obsługuje 4 tryby przekazywania wiadomości: wiadomości punkt do punktu, wiadomości do wielu odbiorców oraz mechanizmy *event* i *query*.

#### Zwykłe wiadomości

Wysyłanie wiadomości punkt do punktu oraz do wielu odbiorców jest zaimplementowane bardzo podobnie. Jak opisałem w sekcji 4.3.3, serwer Multiplexera podejmuje decyzję o dalszym trasowaniu przychodzącej wiadomości na podstawie pól `to` i `type` (oraz rzadko używanego `override_rrules`). Wiadomości punkt do punktu wysyła się poprzez ustawienie pola `to` na identyfikator docelowego odbiorcy lub wypełnienie pola `type` numerem takiego typu wiadomości, który przekazywany jest do jednego odbiorcy. Oczywiście, ten drugi sposób nie pozwala na określenie, kto dokładnie otrzyma daną wiadomość, a zatem jest właściwy dla wysyłania zapytań do serwisów sieciowych w przypadku, gdy nie ma znaczenia, która instancja serwisu obsłuży zapytanie.

Wypełniając pole `type` numerem typu wiadomości, który przekazywany jest do wielu odbiorców, uzyskujemy funkcjonalność rozgłaszania grupowego (ang. *multicast*). Zauważmy, że zgodnie z paradygmatem publikacji-subskrypcji, aplikacja wysyłająca wiadomość *nie musi wiedzieć*, do kogo wiadomość zostanie przekazana ani do ilu odbiorców dotrze. W szczególności możliwe są modyfikacje tablic trasowania powodujące zmiany w liczbie odbiorców ustalonego typu wiadomości.

Dodatkowo można łączyć semantykę wiadomości punkt do punktu i wiadomości przekazywanych do wielu odbiorców. Dla danego typu wiadomości można określić, że wiadomości

tego typu mają być przekazywane do jednego serwisu typu $A$ i wszystkich serwisów typu $B$. Przykładowym zastosowaniem może być zapis do bazy danych poprzez jednego z wielu brokerów (typ $A$), połączony z aktualizacją wszystkich instancji rozproszonej pamięci podręcznej (typ $B$).

Zauważmy, że w przypadku zwykłych wiadomości Multiplexer, podobnie jak inne narzędzia do komunikacji nie replikujące danych, nie daje żadnych gwarancji dostarczenia.

### Mechanizm event

Mechanizm event służy do rozsyłania powiadomień w systemie rozproszonym w przypadku, gdy nadawca nie może lub nie chce zapewnić pewności dostarczenia, ale chce zmaksymalizować jego prawdopodobieństwo. Różni się od zwykłego trasowania wiadomości głównie sposobem wysyłania wiadomości przez aplikację do serwerów Multiplexera. Dla serwera Multiplexera powiadomienie jest zwykłą wiadomością o regułach trasowania określających przekazywanie wiadomości do wielu odbiorców. Dla aplikacji wysyłającej, powiadomienie jest wiadomością pewnego typu, którą wysyła przez *wszystkie* istniejące połączenia z serwerami Multiplexera (lub pewną ich liczbę). Tak więc, jeżeli w systemie uruchomionych jest $N$ serwerów Multiplexera, powiadomienie wysyłane jako event zostaje zwielokrotnione $N$ razy na etapie wysyłania z serwera źródłowego. W tym miejscu szczególnie ważna jest asynchroniczna obsługa operacji wejścia-wyjścia biblioteki klienckiej, dzięki której cały proces *nie trwa* $N$ razy dłużej niż wysyłanie zwykłej wiadomości. Dzięki zwielokrotnieniu wysyłanej wiadomości i przekazywaniu jej różnymi drogami, mamy pewność jej dostarczenia, o ile nie dojdzie do rozległej awarii sieci lub przeciążenia całego systemu. Zauważmy, że w obu tych przypadkach gwarantowanie dostarczenia każdej pojedynczej wiadomości mogłoby uniemożliwić powrót systemu do normalnego działania, a więc nie jest pożądane.

W typowej konfiguracji systemu z $N$ serwerami Multiplexera, każdy komponent systemu podłączony jest do każdego serwera. Ponieważ pojedyncze powiadomienie przekazywane jest niezależnie przez $N$ serwerów Multiplexera, do każdego odbiorcy może być dostarczone wielokrotnie, zwykle właśnie $N$ razy. Aplikacja odbierająca powiadomienie nie chce jednak obsługiwać go wiele razy, zatem konieczna jest *deduplikacja*, czyli usuwanie powtarzających się wiadomości. Ponieważ każda wiadomość jest identyfikowana przez 64-bitową liczbę losową, prawdopodobieństwo kolizji jest zaniedbywalnie małe. Tak więc deduplikacja wymaga pamiętania identyfikatorów ostatnio otrzymanych wiadomości i nie jest kosztowna ani pamięciowo, ani obliczeniowo. Ponieważ serwer Multiplexera nie ma informacji, do ilu ani do których innych serwerów podłączony jest dany klient, deduplikacja może być wykonywana wyłącznie w bibliotece klienckiej. Zauważmy, że w obrębie jednego serwera Multiplexera duplikaty nie tworzą się i zwykle w ogóle nie występują, więc nie ma sensu ich usuwania.

Obecnie biblioteka kliencka nie ma żadnej wbudowanej wiedzy na temat przekazywanych wiadomości, a więc nie odróżnia wiadomości, które mogą wymagać deduplikacji od tych, które nie powinny jej wymagać. W efekcie pamięć przeznaczona na deduplikację jest częściowo zaśmiecona zwykłymi wiadomościami punkt do punktu i wiadomościami do wielu odbiorców nie przekazywanymi poprzez mechanizm event. W przyszłości można rozważyć rozszerzenie protokołu, w którym aplikacja wysyłająca wiadomość poprzez mechanizm event oznacza ją liczbą serwerów, przez które chce ją wysłać, dzięki czemu aplikacja odbierająca wie, że identyfikator powinien być pamiętany dłużej niż w przypadku zwykłych wiadomości.

### Mechanizm query

Mechanizm query można nazwać „rozproszonym RPC", umożliwia bowiem wykonywanie zapytań podobnych do RPC połączone z odpornością na niektóre rodzaje awarii poprzez wyszukiwanie działających powiązań. W celu zapewnienia tych właściwości, query wykorzystuje mechanizm event opisany wcześniej, ale nie używa go w przypadku pełnego powodzenia, by nie obciążać zbytnio sieci i serwerów Multiplexera.

Procedura query składa się z trzech faz i jest używana przez aplikację w celu otrzymania odpowiedzi na wiadomość określonego typu ($T$) i pewnej treści ($M$). W pierwszej fazie zwykła wiadomość typu $T$ o treści $M$ jest wysyłana z wykorzystaniem jednego serwera Multiplexera. Serwer przekazuje ją do serwisu odpowiedniego dla typu $T$, a ten, po obsłużeniu zapytania, przesyła odpowiedź bezpośrednio do nadawcy. O ile nie dojdzie do żadnego błędu, a więc aplikacja inicjująca query dostanie odpowiedź w zadanym czasie, procedura w tym miejscu się kończy przekazując uzyskaną odpowiedź. W przeciwnym przypadku zaszło jedno z następujących zdarzeń:

- brak odpowiedzi w zadanym czasie (zapytanie mogło „zaginąć" z powodu awarii serwera Multiplexera lub serwis je obsługujący nie mógł skończyć procedury obsługi zapytania),
- serwer Multiplexera zgłosi brak pasujących serwisów dla obsługi wiadomości typu $T$ (wiadomość typu `DELIVERY_ERROR`),
- serwis obsługujący wiadomość zgłosi błąd wykonania wysyłając odpowiedź typu `BACKEND_ERROR`.

W każdym z tych przypadków procedura query przechodzi do drugiej fazy, czyli do próby znalezienia działającego serwisu odpowiedniego dla wiadomości typu $T$ wraz z identyfikacją działających połączeń sieciowych pomiędzy poszczególnymi komponentami systemu. W tym celu aplikacja wykonująca query korzysta z mechanizmu event i wysyła wiadomość typu `BACKEND_FOR_PACKET_SEARCH` wraz z liczbą $T$. Wiadomość typu `BACKEND_FOR_PACKET_SEARCH` nie ma określonych reguł trasowania i jest traktowana przez serwer Multiplexera w szczególny sposób. Wiadomość taka, zawierająca liczbę $T$, przekazywana jest zgodnie z *pierwszą* regułą trasowania dla wiadomości typu $T$. Pierwsza reguła dla typu $T$ powinna opisywać serwis obsługujący dane zapytanie, ewentualne następne reguły powinny służyć aktualizacji pamięci podręcznych itp.

Po wysłaniu wiadomości `BACKEND_FOR_PACKET_SEARCH`, procedura query oczekuje na odpowiedź. W przypadku braku odpowiedzi w zadanym czasie lub w przypadku kompletu odpowiedzi typu `DELIVERY_ERROR` wiadomo, że zapytanie nie zostanie już obsłużone i procedura query kończy się błędem. Szczególnym przypadkiem takiego przepływu jest brak w całym systemie odpowiedniego serwisu dla wiadomości typu $T$. Wówczas aplikacja inicjująca query zostanie o tym szybko poinformowana przez serwery Multiplexera i cała procedura nie spowoduje wstrzymania działania aplikacji.

Po otrzymaniu odpowiedzi na `BACKEND_FOR_PACKET_SEARCH` od serwisu $X$, procedura query przechodzi do trzeciej fazy wykonania i ponawia pierwotne zapytanie, wysyłając zwykłą wiadomość z polem `to` ustawionym na identyfikator serwisu $X$. Ponadto serwis $X$ wysyłał odpowiedź na `BACKEND_FOR_PACKET_SEARCH` poprzez serwer Multiplexera, przez który dotarło do niego to zapytanie, a aplikacja wykonująca query wysyła ponowione zapytanie typu $T$ przez ten sam serwer Multiplexera, czyli przez serwer, przez który przyszła odpowiedź od $X$. Ponieważ serwer Multiplexera utrzymuje z każdym klientem co najwyżej jedno aktywne połączenie, tak określone reguły odpowiadania przez ten sam serwer dają gwarancję, że ponowione zapytanie będzie przesyłane poprzez połączenia, o których wiadomo już, że działają. W ten

sposób nie tylko znajduje się działający serwis, zdolny do obsłużenia zapytania typu $T$, ale także identyfikuje się działającą ścieżkę łączącą ten serwis poprzez jeden serwer Multiplexera z aplikacją kliencką. Po ponownym wysłaniu zapytania, procedura query oczekuje na odpowiedź i przekazuje ją lub kończy się błędem, jeżeli odpowiedź nie nadejdzie w wyznaczonym czasie.

W trakcie wykonywania faz drugiej i trzeciej procedury query możliwe jest odebranie odpowiedzi na zapytanie wysłane w fazie pierwszej. Taka sytuacja może zajść na przykład w przypadku większego obciążenia systemu lub gdy procedura obsługi danego zapytania będzie się wykonywać znacznie dłużej, niż przewidziano. W takim przypadku procedura query kończy się przekazując uzyskaną odpowiedź, a późniejsze odpowiedzi na `BACKEND_FOR_PACKET_SEARCH` są ignorowane.

Zauważmy, że nie jest możliwe jednoczesne zapewnienie otrzymania odpowiedzi i ograniczenie czasu wykonania procedury query, ponieważ nie jest możliwe przewidzenie przerw w działaniu całej sieci lub innych poważnych problemów infrastrukturalnych. Multiplexer powstał jako narzędzie do obsługi portalu internetowego, gdzie krótki czas wykonania zadania jest równie ważny, jak samo wykonanie. W związku z tym, procedura query, gwarantująca otrzymanie odpowiedzi, ale nie ograniczająca czasu wykonania, nie zaspokajałaby potrzeb, które doprowadziły do stworzenia Multiplexera.

Jednakże, tak zaprojektowany mechanizm query umożliwia przezroczyste radzenie sobie w przypadku „zwykłych" awarii systemu, błędów sieci lub błędów procesów. Daje więc on najmocniejsze praktyczne gwarancje wykonania i optymalnie zaspokaja zapotrzebowanie portalu Azouk.

### 4.4.3. Duża przepustowość

Serwer Multiplexera napisany jest jako jednowątkowa aplikacja, asynchronicznie obsługująca operacje wejścia-wyjścia. Dzięki optymalizacji kodu w trakcie kompilacji pod kątem jednowątkowego wykonania, zminimalizowany został koszt związany z zarządzaniem pamięcią i zwalnianiem nieużywanych obiektów, ponieważ nie są do tego potrzebne żadne mechanizmy synchronizacji. Tak więc najczęściej przepustowość sieci, a nie serwerów Multiplexera, ogranicza wydajność systemu.

Jednakże w przypadku przesyłania wielu wiadomości w obrębie jednego serwera fizycznego lub przesyłania wielu małych wiadomości, powodujących małe wykorzystanie sieci, ale normalne wykorzystanie serwera Multiplexera, jego przepustowość może się stać „wąskim gardłem". Ponieważ poszczególne serwery są niezależne, aby rozwiązać ten problem wystarczy zwiększyć ich liczbę i w ten sposób zmniejszyć liczbę wiadomości, jaką każdy z nich musi obsłużyć w jednostce czasu.

### 4.4.4. Model publikacji-subskrypcji

Jak wspomniałem w sekcji 1.1.5, w systemie rozproszonym składającym się z luźno powiązanych komponentów bardzo ważne jest rozsyłanie powiadomień o zmianach (powiadomienie o zmianie odpowiada *publikacji* w modelu publikacji-subskrypcji). W serwisie Azouk dynamiczny rozwój systemu stwarza problem z zarządzaniem konfiguracją rozsyłania powiadomień.

Multiplexer rozwiązuje ten problem dzięki scentralizowanej konfiguracji rozsyłania powiadomień. Reguły trasowania dla wiadomości informujących o zdarzeniach w systemie można

dowolnie zmieniać w pliku konfiguracyjnym serwerów, bez konieczności modyfikacji czy nawet restartu pozostałych komponentów systemu.

Zauważmy, że powiadomienia, w odróżnieniu od żądań używających mechanizmu query, przesyłane są z wykorzystaniem mechanizmu event. Naturalnie powiadomienia są wiadomościami *nie wymagającymi* odpowiedzi, a dokładniej nadawca nie oczekuje na odpowiedź. Nigdy w związku z samą zmianą subskrypcji coś, co było żądaniem, nie stanie się powiadomieniem, ani coś, co było powiadomieniem – żądaniem. Tak więc, zgodnie z założeniami modelu publikacji-subskrypcji, zmiana subskrypcji nigdy nie wymaga zmian w kodzie odpowiedzialnym za publikację (generującym powiadomienie).

### 4.4.5. Zbieranie informacji diagnostycznych

W każdym systemie, który posiada komponenty działające bez interakcji z użytkownikiem, bardzo ważny jest podsystem logowania. Zadaniem logowania jest zbieranie i zapisywanie informacji diagnostycznych i statystycznych, które mogą posłużyć do analizowania błędów i wszelkich nieoczekiwanych zachowań, mierzenia wydajności lub zwykłej analizy zachowania systemu. Zwykle programy zapisują logi w podkatalogu `/var/log` na serwerze, na którym są uruchomione. Konwencja ta jest bardzo wygodna w sytuacji, w której wiemy, na którym serwerze wykonywał się interesujący nas proces. W systemach rozproszonych możliwe są sytuacje, gdy

- nie wiemy, na której maszynie wykonywał się interesujący nas proces lub
- nie wiemy nawet, który proces nas interesuje.

W takich sytuacjach nadzwyczaj pożądana jest możliwość centralnego zbierania i przeszukiwania logów. Najprostszym rozwiązaniem wydaje się nakładka na standardowy system logowania łącząca pliki pochodzące z katalogów `/var/log` z różnych serwerów. Takie rozwiązanie ma tę wadę, że łącząc logi tracimy informację o tym, z której maszyny pochodzą. Standardowe narzędzia do logowania nie zachowują informacji o serwerze, na którym logi są generowane – oczywiście można taką informację dołączać, trzeba ją wówczas dodatkowo przetwarzać. Azouk Logging oferuje jednak rozwiązanie bardziej uniwersalne, pozwalające w przejrzysty sposób systematyzować i grupować logi.

W systemie rozproszonym serwery, działające na nich procesy oraz usługi dostarczane przez te procesy są obiektami stanowiącymi potencjalne źródła logów. Strukturę tych obiektów można odwzorować w postaci drzewa etykietowanego ich nazwami. Rozpatrzmy maszynę `violet` oraz działające na niej procesy `mxserver`, będący serwerem Multiplexera i `website`, serwujący strony WWW. Na serwerze `violet` procesy te mogą tworzyć przykładowo następujące źródła logów (por. rys. 4.10):

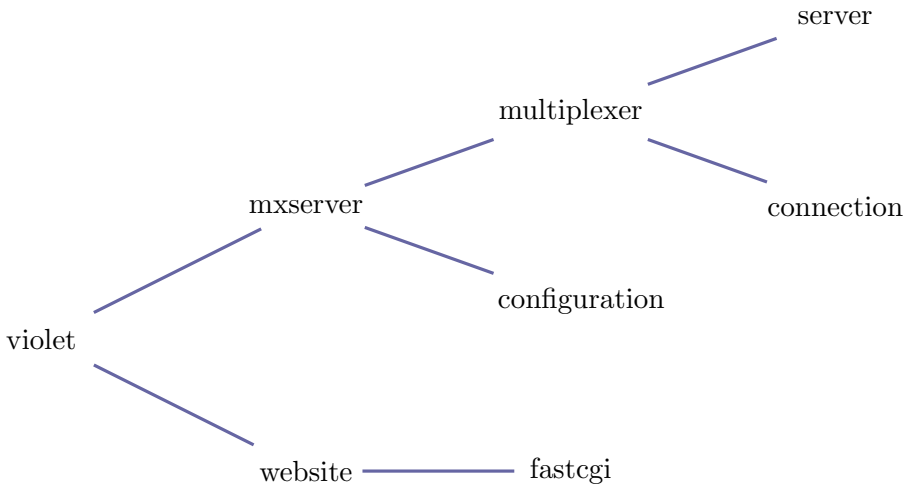
**violet.mxserver.multiplexer** dla zdarzeń ogólnych wygenerowanych przez bibliotekę Multiplexera w procesie `mxserver`,

**violet.mxserver.multiplexer.server** dla zdarzeń wygenerowanych przez obiekt będący właściwym serwerem Multiplexera w procesie `mxserver`,

**violet.mxserver.multiplexer.connection** dla zdarzeń wygenerowanych przez obiekty obsługujące poszczególne połączenia w procesie `mxserver`,

**violet.mxserver.configuration** dla zdarzeń wygenerowanych przez moduł konfiguracji w procesie `mxserver`,

**violet.website.fastcgi** dla zdarzeń wygenerowanych w procesie `website` przez moduł FastCGI.




**Rysunek 4.10:** Przykład źródeł logowania w Azouk Logging.

W tak skonstruowanym drzewie obiektów etykietowanie logów według ścieżki pozwala na zastosowanie tych samych metod filtrowania i grupowana logów na różnych poziomach zagłębienia. Na przykład w interfejsie użytkownika przeglądarki logów można w jednym miejscu prezentować logi pochodzące z całego serwera lub tylko z jednej biblioteki w jednym procesie. W naszym przykładzie mogłyby to być logi z całego serwera `violet` – wszystkie logi z drzewa o korzeniu `violet` – lub tylko z biblioteki Multiplexera w procesie `mxserver` – wszystkie logi z poddrzewa `violet.mxserver.multiplexer`.

### Interfejs

Generowanie logów w trakcie wykonania programu zawsze jest kosztowną operacją, za to ważne jest, by instrukcje logowania nie powodujące generowania logów, były wykonywane jak najszybciej. W tym celu *sprawdzanie*, czy dana informacja ma być logowana, musi być wykonywane jak najwcześniej, najlepiej przed jakimkolwiek formatowaniem napisów czy nawet zwykłą konkatenacją. Co prawda w językach programowania korzystających z preprocesora, takich jak C i C++, możliwe jest zupełne usunięcie zbędnego kodu logującego w trakcie kompilacji, nie jest to jednak możliwe w językach takich jak Java czy Python, które preprocesora nie mają. Ponadto zupełne usunięcie kodu logującego zazwyczaj nie jest pożądane – w systemie rozproszonym możemy chcieć *włączyć* logowanie w poszczególnych komponentach bez ich rekompilacji, a tym bardziej bez restartu.

API biblioteki Azouk Logging dla C++ i Pythona zostało zaprojektowane właśnie z myślą o minimalizacji kosztu instrukcji logowania, które nie powodują faktycznego generowania logów. Mimo takiego założenia, udało się osiągnąć interfejs biblioteki pozwalający na elastyczne budowanie logowanych informacji.

**Interfejs dla Pythona** W dynamicznym języku, jakim jest Python, łatwo stworzyć API, które jest elastyczne i nie wymaga dodatkowych obliczeń przed sprawdzeniem, czy log zostanie wygenerowany. W uproszczeniu, parametr `text` wywołania Azouk Logging może być napisem do wypisania lub funkcją przekazującą taki napis. Dzięki Pythonowym lambda-wyrażeniom, można w ten zwięzły sposób opóźnić formatowanie napisu do momentu, gdy wiadomo, że zostanie on wykorzystany.

**Interfejs dla C++** W języku kompilowanym dużo trudniej osiągnąć elastyczne API. Programując w C++ czasem liczy się nawet liczbę wywoływanych funkcji, a więc instrukcje logowania nie powinny wykonywać zbyt wielu wywołań przed sprawdzeniem, że dana informacja ostatecznie zostanie zapisana. Azouk Logging osiąga to za pomocą programowania w preprocesorze, wykorzystującego bibliotekę Boost Preprocessor (por. sekcja 4.2.3). Przykładowe wywołanie Azouk Logging może wyglądać tak:

```
AZOUK_LOG(WARNING, HIGHVERBOSITY,
        CTX("multiplexer.connection")
        TEXT("connection with peer " + str(peer_id) + " was lost"));
```

Makro `AZOUK_LOG(level, verbosity, args...)` rozwija się do kodu zaczynającego się od sprawdzenia, czy wiadomość o ważności `level` i częstości występowania `verbosity` powinna zostać zapisana. Dopiero po tym sprawdzeniu parametr `args` jest interpretowany. `args` składa się z serii „znaczników" (w przykładzie są to `CTX` i `TEXT`, mogą być argumentowe i bezargumentowe). Tak więc kod wewnątrz `TEXT(...)` zostanie wykonany tylko wtedy, gdy jest to potrzebne, ale dzięki zastosowaniu makr preprocesora ma on dostęp do wszystkich zmiennych ze środowiska otaczającego wywołanie.

Przytoczony wcześniej opis jest, z konieczności, pewnym uproszczeniem. W rzeczywistości parametr `args` interpretowany jest wiele razy. To umożliwia m.in. uzależnienie decyzji o logowaniu od kontekstu wywołania (znacznik `CTX`), zdefiniowanie znacznika `MUSTLOG`, wymuszającego wygenerowanie logu, znacznika `DATA`, umożliwiającego dołączenie do wygenerowanego logu dowolnej wiadomości Google Protocol Buffers (instancjonowanej i inicjowanej tylko wtedy, gdy informacja ma być ostatecznie zapisana).

### Przesyłanie logów

W Multiplexerze logi generowane przez różne procesy traktowane są na równi z innymi danymi. Przesyłane logów wykonywane jest przez dedykowane procesy za pośrednictwem serwerów Multiplexera. Wydzielenie przesyłania logów poza obręb procesu, który je generuje, jest konieczne, by zapewnić, że logi nie będę gubione w przypadku problemów z siecią lub na przykład nagłego zakończenia procesu.

Ponieważ serwery Multiplexera w trakcie obsługi przychodzących wiadomości także mogą generować logi, a te z kolei mogą być od razu przesyłane do procesów je zbierających, ważne jest zagwarantowanie, że obsługa całego procesu logowania nie powoduje stałego wzrostu liczby wymienianych wiadomości w systemie. Doprowadziłoby to do stałego „szumu" lub nawet „eksplozji" ruchu w systemie. W Multiplexerze rozwiązano ten problem na dwa sposoby. Po pierwsze, zalogowane zdarzenia nie są przesyłane pojedynczo, tylko w paczkach po kilkanaście-kilkadziesiąt logów. Po drugie, wiadomości z paczkami logów są specjalnie oznaczone tak, by same nie powodowały tworzenia nowych logów. Przy włączonym generowaniu logów na odpowiednim poziomie, serwer Multiplexera, rozpoznając takie oznaczenia, co najwyżej wypisuje informacje diagnostyczne na konsolę w formacie czytelnym dla człowieka, ale nigdy nie zapisuje ich do pliku ani nie przekazuje do procesu dedykowanego do przesyłania logów.

Dzięki zbieraniu na bieżąco logów pochodzących od różnych komponentów systemu, Azouk Logging jest ważną częścią Multiplexera. Umożliwia szybką diagnostykę systemu rozproszonego i reagowanie na nieprawidłowe sytuacje. Gotowe programy pomocnicze – do przesyłania i do zbierania logów – pozwalają na szybką konfigurację środowiska i sprawiają, że Multiplexer doskonale spełnia wymaganie opisane w sekcji 1.1.7.

## 4.5. Podsumowanie

Multiplexer, stworzone przeze mnie narzędzie do komunikacji, spełnia wszystkie wymagania stawiane mu przez środowisko portalu Azouk, na którego potrzeby powstał. Naturalnie Multiplexer jest odpowiednim narzędziem do komunikacji także dla innych portali internetowych, gdzie od transakcyjnego działania ważniejsze są szybkość, wydajność i odporność na awarie.

Dzięki użyciu zewnętrznych bibliotek Boost, Asio i Protocol Buffers, możliwe było osiągnięcie wydajnej, asynchronicznej komunikacji z użyciem przenośnego protokołu. Zastosowany protokół jest nie tylko przenośny, ale i rozszerzalny – wybrane możliwe rozszerzenia są opisane w sekcji 6.1.

Multiplexer powstał na potrzeby portalu Azouk, z powodu braku dostępnych narzędzi do komunikacji posiadających pożądane cechy. Spread najlepiej spośród istniejących narzędzi spełnia wymagania stawiane przez portal (por. sekcja 3.2.4). W następnym rozdziale uzasadniam sensowność stworzenia nowego narzędzia do komunikacji mimo dużej liczby istniejących rozwiązań oraz porównuję wybrane aspekty funkcjonalności i wydajności Spreada i Multiplexera.

# Rozdział 5

# Multiplexer a Spread

Z analizy dostępnych narzędzi do komunikacji, dokonanej w rozdziale 3, wynika, że Spread (por. sekcja 3.2.4) najlepiej spośród nich pasuje do wymagań stawianych Multiplexerowi. Tak więc w niniejszym rozdziale porównuję Multiplexer wyłącznie ze Spreadem. Porównanie obejmuje aspekty modelu komunikacji, funkcjonalności, gwarancji dostarczenia oraz wydajności i odporności na awarie.

## 5.1. Porównanie funkcjonalne

### 5.1.1. Model komunikacji

#### Spread

Spread jest narzędziem do komunikacji grupowej [24], realizującym model *Extended Virtual Synchrony* [18]. W komunikacji grupowej podstawowymi pojęciami są *grupa*, *przynależność do grupy* i *wysyłanie wiadomości do grupy*. Extended Virtual Synchrony jest modelem poprawności dostarczania wiadomości w komunikacji grupowej, w którym wiadomości dostarczane są w określonym porządku, wspólnym dla wszystkich uczestników komunikacji i w taki sposób, że nadawca może określić, którzy odbiorcy są dostępni w momencie dostarczenia wiadomości.

Model komunikacji grupowej przewiduje, że aplikacja korzystająca ze Spreada zarządza przynależnością do grup. W szczególności musi wiedzieć, do jakich grup dołączyć i co robić z wiadomościami o dołączaniu innych. Wiadomości wysyłane przez Spreada muszą być kierowane do konkretnej grupy odbiorców. Dodatkowo, każdy uczestnik komunikacji powinien posiadać swoją prywatną grupę, której jest jedynym członkiem, aby umożliwić wysyłanie wiadomości „grupowych" do konkretnego, pojedynczego odbiorcy.

#### Multiplexer

Multiplexer jest narzędziem do komunikacji zorientowanym na wiadomości. W Multiplexerze nie ma pojęcia *uczestnictwa w grupie* czy samej *grupy*, a wiadomości przesyłane przez serwer Multiplexera oznaczone są typem wiadomości, na podstawie którego serwer decyduje, komu dostarczyć wiadomość. Taki model komunikacji przewiduje tylko, że aplikacja korzystająca z Multiplexera poda identyfikator swojego typu, by serwery Multiplexera, do których się podłączy, wiedziały, które wiadomości do niej kierować.

Multiplexer ukierunkowany jest na zadania aplikacyjne, a więc umożliwia przesyłanie wiadomości, odpytywanie usług, wysyłanie powiadomień. Dzięki centralnej bazie reguł przesyłania wiadomości, Multiplexer doskonale realizuje model publikacji-subskrypcji.

### 5.1.2. Architektura

#### Spread

Instalacja Spreada składa się ze zbioru demonów i zbioru klientów do nich podłączonych. Klient identyfikowany jest przez swoje połączenie z demonem, a więc klient podłączony do dwóch demonów widziany jest w systemie jako dwóch niezależnych uczestników komunikacji. Aplikacja korzystająca ze Spreada może, w celu podniesienia niezawodności, samodzielnie zarządzać większą liczbą połączeń lub zdać się w tym zakresie na usługi demona, do którego się podłącza. To drugie rozwiązanie jest zalecane, ale przy jego stosowaniu odporność na awarie sieci wymaga instalowania demonów Spreada na każdej maszynie fizycznej i podłączania klientów do demonów znajdujących się na ten samej maszynie.

Do komunikacji między demonami w sieci lokalnej Spread używa krążącego żetonu. Awaria demona aktualnie posiadającego żeton powoduje przerwę w działaniu systemu w celu jego automatycznej rekonfiguracji i odtworzenia żetonu. Ponadto taka konfiguracja nie jest skalowalna ze względu na rosnącą liczbę maszyn fizycznych w systemie.

#### Multiplexer

Instalacja Multiplexera składa się z jednej lub więcej (typowo trzech) instancji serwerów Multiplexera, do których podłączają się klienci. Każdy klient podłącza się do wszystkich znanych serwerów Multiplexera, a serwery nie łączą się między sobą. Taka architektura zapewnia, że zerwanie jednego połączenia klient-serwer nie powoduje przerwy w komunikacji, a awaria jednego z serwerów nie ma wpływu na pracę pozostałych.

Naturalną konsekwencją niezależności serwerów Multiplexera jest skalowalność całego systemu. Przy dużej liczbie zwykłych wiadomości przesyłanych w systemie wystarczy dostawienie nowego serwera Multiplexera, by zmniejszyć obciążenie poszczególnych serwerów.

### 5.1.3. Funkcjonalność i gwarancje

Zarówno Spread, jak i Multiplexer umożliwiają wysyłanie wiadomości składających się z dowolnych ciągów bajtów. Różnią je natomiast gwarancje dostarczenia przesyłanych wiadomości oraz sposób opisu i możliwości wykorzystania metadanych.

#### Spread

API Spreada jest proste i nie ulegało większym zmianom od kilku lat. Umożliwia ono wysyłanie wiadomości grupowych o określonym poziomie gwarancji dostarczenia. Wyróżnione są następujące poziomy gwarancji (wymienione w kolejności od najsłabszego do najsilniejszego):

**unreliable** przesyłana wiadomość może zostać zgubiona; brak gwarancji,

**reliable** przesyłana wiadomość zostanie dostarczona do wszystkich członków grupy nawet w przypadku awarii sieci,

**FIFO** podobnie jak *reliable*, ale dodatkowo wszystkie wiadomości poziomu *FIFO* lub wyższego, pochodzące od jednego nadawcy, będą dostarczane odbiorcom w porządku wysłania,

**causal** podobnie jak *reliable*, ale wszystkie wiadomości poziomu *causal* lub wyższego, od wszystkich nadawców, będą do każdego odbiorcy dostarczone zgodnie z częściowym porządkiem wyznaczonym przez zegary Lamporta (ang. *causal ordering*),

**agreed** podobnie jak *causal*, ale częściowy porządek wyznaczony przez zegary Lamporta zastąpiony jest liniowym, ustalonym poprzez głosowanie przy użyciu ID nadawcy,

**safe** podobnie jak *agreed*, ale wiadomość może zostać dostarczona przez demon Spreada do klienta tylko wtedy, gdy wszystkie inne demony mają już tę wiadomość lub demon Spreada zidentyfikuje podział sieci.

Spread umożliwia wysyłanie wiadomości do grup, ale nie dostarcza funkcjonalności kolejek komunikatów. Aby móc zastosować Spreada jako platformę do dostarczania usług dla systemu rozproszonego, konieczne jest zaimplementowanie tej funkcjonalności poprzez zastosowanie menedżera zadań lub takie zsynchronizowanie serwisów dostarczających usługi, by każde zadanie było obsługiwane przez dokładnie jeden z nich (jest to możliwe, gdy zadania są wysyłane jako wiadomości z gwarancją dostarczenia *agreed* lub *safe*).

#### Multiplexer

W Multiplexerze gwarancje dostarczenia realizowane są przez bibliotekę kliencką, a nie przez zewnętrzną usługę, np. serwery lub demony. Multiplexer nie zapewnia określonego porządku wiadomości, chyba że są one przesyłane przez jeden serwer – wówczas uporządkowanie wiadomości jest takie jak *FIFO* Spreada. Z drugiej strony, gwarancje dostarczenia nie zależą od działania serwerów, co ułatwia skalowalność oraz zapewnia takie same gwarancje niezależnie od przejściowych problemów z siecią.

Multiplexer wspiera przesyłanie wiadomości zarówno w sposób podobny do wiadomości grupowych, jak i podobny do kolejek komunikatów. Dzięki temu Multiplexer świetnie pasuje jako narzędzie do wysyłania powiadomień o zmianach oraz jako platforma do dostarczania usług dla systemu rozproszonego.

### 5.1.4. Metadane

#### Spread

Spread do komunikacji pomiędzy klientem a demonem używa binarnego protokołu o stałym układzie danych. Umożliwia to efektywne przesyłanie takich metadanych jak typ wiadomości, rodzaj gwarancji dostarczenia, nadawca wiadomości, ale nie pozwala na na zmianę protokołu bez równoczesnej aktualizacji wszystkich uczestników komunikacji. Ponadto metadane przekazywane przez Spreada są trudno dostępne poprzez jego C API i dodanie nowego ich typu wymagałoby zmiany API, a więc również zmiany aplikacji z niego korzystających.

#### Multiplexer

Multiplexer przesyła wiadomości w rozszerzalnym formacie kodowania Google Protocol Buffers. Dzięki temu, poza zasadniczymi wiadomościami, a czasami zamiast nich, aplikacje mogą

przesyłać między sobą metadane zapisane razem z innymi danymi protokołu. Ta funkcjonalność umożliwia płynną ewolucję protokołu i wymienianych danych nawet wtedy, gdy aplikacje nie są gotowe na zmianę formatu zasadniczych wiadomości. Dodanie nowego typu metadanych nie wymaga żadnej modyfikacji serwera ani tych aplikacji, które nie są zainteresowane odczytywaniem tych nowych danych. Obecnie Multiplexer przekazuje wraz z każdą wiadomością następujące informacje (por. sekcja 4.3.2): ID wiadomości, ID nadawcy, typ wiadomości, czas wysłania, workflow itd.

## 5.2. Wydajność i stabilność

W tym rozdziale porównuję Multiplexera i Spreada pod kątem wydajności w typowych zastosowaniach biznesowych, takich jak pośredniczenie w dostępie do usług i rozsyłanie powiadomień.

### 5.2.1. Test: odpytywanie usług, bez użycia sieci

#### Środowisko

Test został przeprowadzony na komputerze z dwurdzeniowym procesorem Intel Core 2 Duo T7500 o maksymalnej częstotliwości taktowania 2.2 GHz i pamięci podręcznej L2 wielkości 4 MB, osobno dla każdego rdzenia. System testowy posiadał 4 GB pamięci głównej, a test został tak zaprojektowany i przeprowadzony, by nie wymagał używania partycji wymiany ani intensywnych odczytów z dysku.

Do porównania użyłem Spreada w wersji 4.0.0, dystrybuowanej w postaci binarnej oraz wersji Multiplexera zmodyfikowanej jedynie tak, by błędne uruchomienia szybciej się kończyły.

Każda konfiguracja testowa była uruchomiona 3 razy, a wyniki zebrane z poszczególnych uruchomień zostały uśrednione.

#### Scenariusz

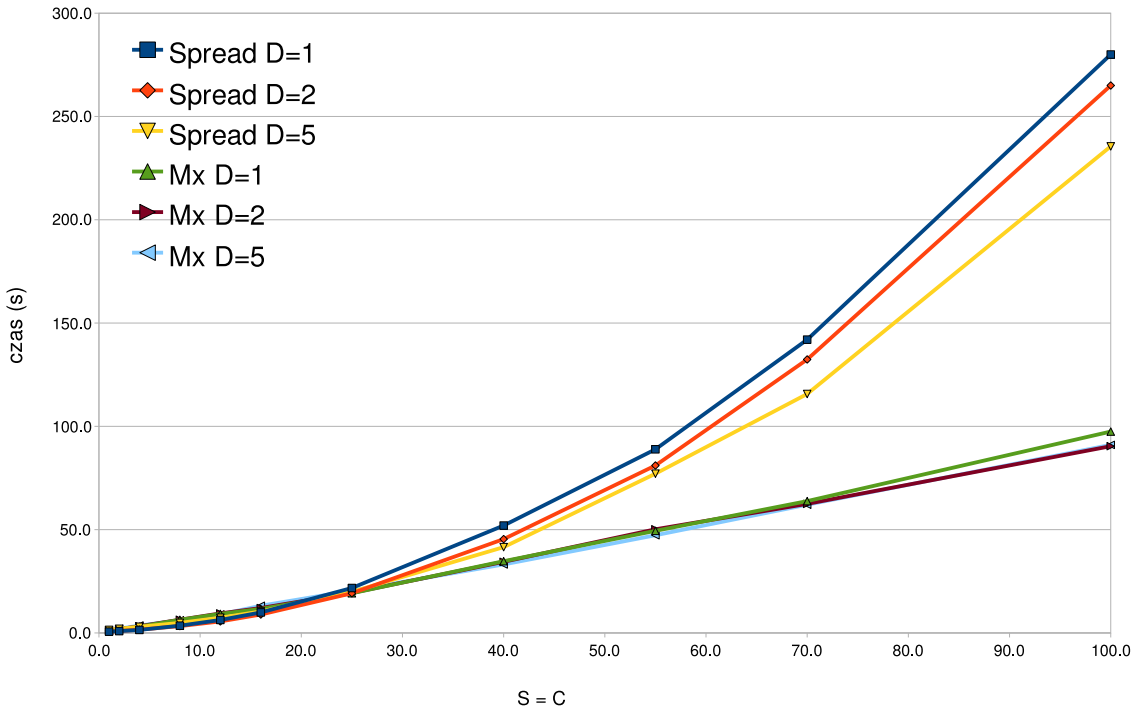
W tym teście uruchamianych było $D$ demonów (Spreada lub Multiplexera, odpowiednio), $S$ usług typu echo i $C$ procesów klienckich odpytujących te usługi, dla różnych parametrów $D$, $S$, $C$. Każdy z procesów klienckich wykonywał $T = 5000$ żądań, na każde z nich odpowiadała dokładnie jedna z usług wysyłając z powrotem te same dane, które przesłał klient, wielkości 45 bajtów. Wynikiem testu był czas działania procesów klienckich.

W przypadku Spreada każda z usług i każdy z procesów klienckich łączył się z losowo wybranym demonem Spreada (biblioteka Spreada nie obsługuje połączeń z wieloma demonami naraz). Żądania od procesów klienckich przesyłane były do jednej wspólnej grupy usług z poziomem gwarancji *agreed* tak, by każda usługa wiedziała, na które żądania powinna odpowiadać. Odpowiedzi do klientów przesyłane były z poziomem gwarancji *reliable* do prywatnej grupy danego klienta.

W przypadku Multiplexera każda z usług i każdy z procesów klienckich łączył się ze wszystkimi uruchomionymi serwerami. Każde żądanie przesyłane było przez jeden serwer Multiplexera i trasowane przez niego do jednej z podłączonych usług.

### Wyniki

Rysunek 5.1 przedstawia wyniki testu dla równej liczby procesów klienckich i instancji usługi ($S = C$). Na wykresie widać liniową zależność czasu trwania testu od liczby uruchomionych procesów w przypadku Multiplexera oraz w przybliżeniu kwadratową[1] – w przypadku Spreada. Zauważmy, że Spread umożliwia jedynie grupowe rozsyłanie wiadomości, a więc proces kliencki nie może wysłać żądania do dowolnej jednej *instancji* usługi, o ile nie posiada listy wszystkich dostępnych instancji. Oczywiście każdy z procesów klienckich mógłby taką listę konstruować raz, przy starcie, wysyłając wiadomość do wszystkich usług i zapamiętując te, które odpowiedziały. Jednakże takie rozwiązanie nie sprawdza się w realnych zastosowaniach, gdzie cały system musi być otwarty na podłączanie nowych usług i dynamicznie reagować na odłączanie się istniejących. Tak więc Spread musi przesłać $T * (C + S * C)$ komunikatów zamiast $T * 2 * C$. Właśnie ta zależność $\mathcal{O}(S * C)$ liczby przesłanych pakietów decyduje o zdecydowanie gorszych wynikach Spreada w tym teście.




**Rysunek 5.1:** Usługa echo, bez sieci

Mimo niedużej liczby powtórzeń testu dla każdej konfiguracji testowej, średnia procentowego odchylenia standardowego (tj. stosunku odchylenia standardowego serii do średniej wyników serii) była niewielka. Dokładniej, niech $X$ będzie dyskretną zmienną losową o równomiernym rozkładzie i o wartościach będących wynikami poszczególnych uruchomień testu dla pojedynczej konfiguracji. Wówczas niech $\sigma_{\%}(X)$ będzie procentowym odchyleniem standardowym pojedynczej konfiguracji:

$$\sigma_{\%}(X) = \frac{\sigma(X)}{E(X)} = \frac{\sqrt{E(X^2) - (E(X))^2}}{E(X)}.$$

Średnia $\sigma_{\%}$ po różnych konfiguracjach testowych, oznaczana $\overline{\sigma_{\%}}$, wyniosła w tym teście 3.13%, a więc uzyskane wyniki są wiarygodne.

[1] Wykres dla Spreada przy $D = 1$ jest niemal identyczny z wykresem funkcji $f(x) = 0.025 * x^2 + 0.3 * x$.

### 5.2.2. Test: odpytywanie usług

Ten test jest analogiczny do testu 5.2.1 z tą tylko różnicą, że tutaj poszczególne procesy komunikowały się ze sobą poprzez sieć LAN.

#### Środowisko

Test został przeprowadzony na 15 komputerach połączonych gigabitową siecią lokalną. Każdy komputer był wyposażony w dwurdzeniowy procesor Intel Core 2 Duo E8400 o maksymalnej częstotliwości taktowania 3 GHz i pamięci podręcznej L2 wielkości 6 MB, osobno dla każdego rdzenia oraz 4 GB pamięci głównej.

Do porównania użyłem demona Spreada w wersji 4.0.0, dystrybuowanej w postaci źródeł i zmodyfikowanej tak, by demon dopuszczał 1048576 wiadomości oczekujących dla pojedynczego klienta przed zerwaniem jego połączenia (domyślny limit to 1000). Demon Multiplexera nie wymagał analogicznej modyfikacji, ponieważ dopuszczalna liczba oczekujących wiadomości jest konfigurowalna podczas uruchomienia.

Każda konfiguracja testowa była uruchomiona 3 razy, a wyniki zebrane z poszczególnych uruchomień zostały uśrednione.

#### Scenariusz

Scenariusz tego testu był taki sam jak w przypadku testu 5.2.1 z tą różnicą, że kolejne procesy z jednej konfiguracji testowej były uruchamiane cyklicznie na kolejnych komputerach testowych.

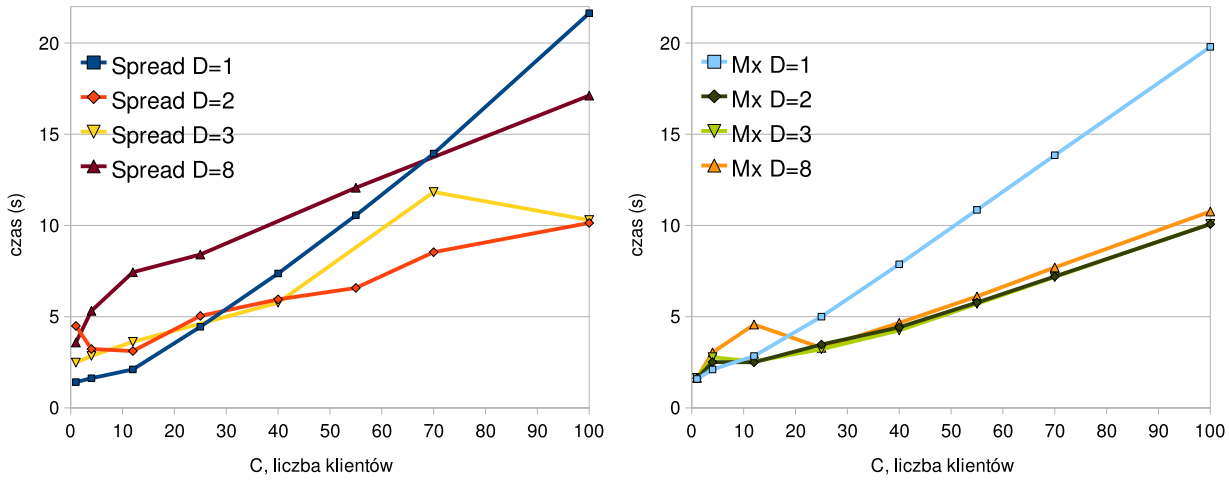
#### Wyniki

Rysunek 5.2 przedstawia wyniki testu dla $S = 1$ ($\overline{\sigma_{\%}} = 3.64\%$). Zauważmy, że w tej konfiguracji Spread i Multiplexer mają do dostarczenia dokładnie tyle samo wiadomości – Multiplexer nie jest faworyzowany przez fakt, że Spread musi dostarczyć $\mathcal{O}(S * C)$ wiadomości. Rysunek 5.2b pokazuje, że przy jednej instancji usługi i jednym demonie Multiplexera wydajność systemu jest ograniczona przez demona – dodanie drugiego demona zmniejsza czas wykonania testu. Jest to zrozumiałe, ponieważ demon obsługuje wiele połączeń naraz i jego zadanie (trasowanie pakietów) jest bardziej skomplikowane niż zadanie usługi (echo). Gdy uruchomionych jest więcej demonów, wydajność ograniczona jest przez pojedynczą instancję usługi, a więc wyniki Multiplexera dla $D = 2, 3, 5$ i 8 są bardzo podobne (wyniki dla $D = 5$ nie zostały przedstawione na wykresie dla zachowania czytelności).

Rysunek 5.3 przedstawia wyniki testu dla $S = 40$ ($\overline{\sigma_{\%}} = 3.44\%$). Ogólnie, jak wykazały inne testy, których wyników nie przytaczam tutaj dla zwięzłości, im więcej jest uruchomionych instancji usługi ($S$), tym wyniki Spreada są gorsze a Multiplexera – lepsze.

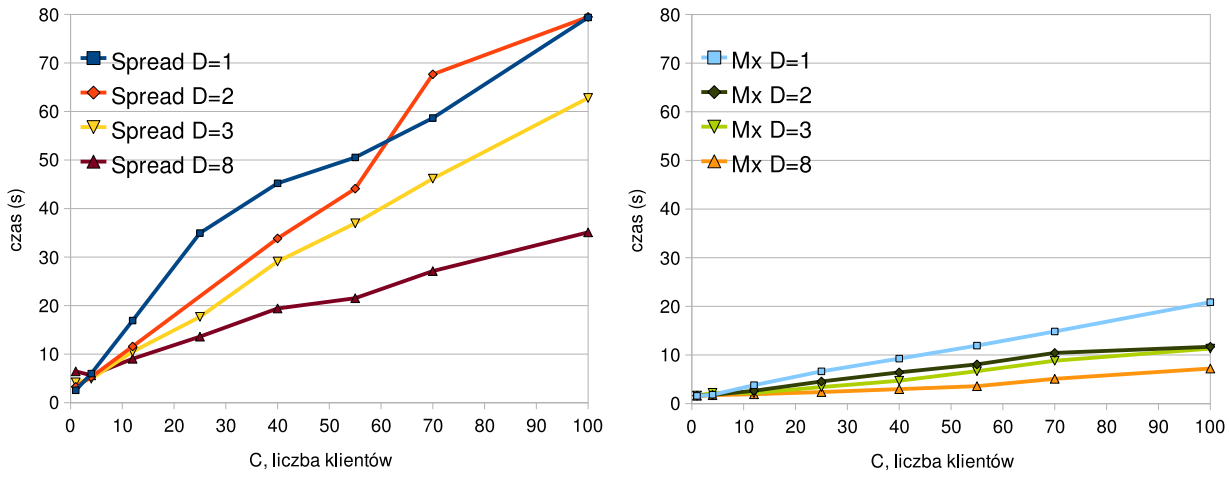
### 5.2.3. Test: rozsyłanie małych powiadomień

Rozsyłanie powiadomień jest drugim ważnym zadaniem, jakie ma spełniać Multiplexer, zaraz po pośredniczeniu w dostępie do usług. Jednakże rozsyłanie powiadomień, czy też ogólnie wiadomości grupowych, jest podstawowym celem Spreada, więc konkurowanie z nim na tym polu jest trudne.




**(a)** Spread **(b)** Multiplexer

**Rysunek 5.2:** Usługa echo, 1 instancja




**(a)** Spread **(b)** Multiplexer

**Rysunek 5.3:** Usługa echo, 40 instancji

### Środowisko

Środowisko tego testu, podobnie jak środowisko testów opisanych dalej, jest tym samym, w którym został przeprowadzony test 5.2.2.

### Scenariusz

W tym teście uruchamianych było $D$ demonów (Spreada lub Multiplexera, odpowiednio), $S$ odbiorców powiadomień i $C$ procesów wysyłających powiadomienia, dla różnych parametrów $D$, $S$, $C$. Każdy z procesów wysyłających powiadomienia nadawał $T = 5000$ wiadomości, z których każda miała dotrzeć do *wszystkich* odbiorców. Każda wiadomość była wielkości 45 bajtów. Na zakończenie każdy z odbiorców wysyłał pojedyncze potwierdzenie do nadawców sygnalizujące, że otrzymał wszystkie wiadomości. Każdy proces nadający wiadomości kończył

się po odebraniu wszystkich potwierdzeń. Wynikiem testu był czas działania nadawców.

Podobnie jak w poprzednich testach, w przypadku Spreada każdy odbiorca i nadawca łączył się z losowo wybranym demonem. Wszystkie wiadomości i końcowe potwierdzenia wysyłane były z poziomem gwarancji *reliable* do jednej wspólnej grupy odbiorców lub nadawców, odpowiednio.

W przypadku Multiplexera każdy odbiorca i nadawca łączył się ze wszystkimi uruchomionymi demonami. Każda wiadomość przesyłana była przez jeden demon Multiplexera i trasowana przez niego do wszystkich podłączonych odbiorców lub nadawców, odpowiednio.

#### Wyniki

Rysunki 5.4a i 5.4b przedstawiają wyniki testu dla liczby odbiorców $S = 50$ ($\overline{\sigma_{\%}} = 8.65\%$). Zauważmy, że we wszystkich konfiguracjach oprócz konfiguracji z jednym demonem ($D = 1$), Multiplexer osiąga wyniki wyraźnie lepsze niż Spread. Podobnie kształtują się wyniki dla $S = 70$ ($\overline{\sigma_{\%}} = 7.33\%$), przedstawione na rysunkach 5.4c i 5.4d. Ponadto rysunki 5.4b i 5.4d pokazują, że Multiplexer bardzo dobrze się skaluje – podwajanie liczby demonów niemal dwukrotnie skraca czas wykonania testu.

### 5.2.4. Test: rozsyłanie dużych powiadomień

Ten test jest analogiczny do testu 5.2.3 z tą tylko różnicą, że tutaj nadawcy wysyłają wiadomości rozmiaru 1024 bajtów. Ze względu na zbyt długi czas wykonania testu, konfiguracje z jednym demonem ($D = 1$) nie zostały uruchomione dostatecznej liczby razy i odpowiednie wyniki nie zostały przedstawione na załączonych wykresach.

#### Wyniki

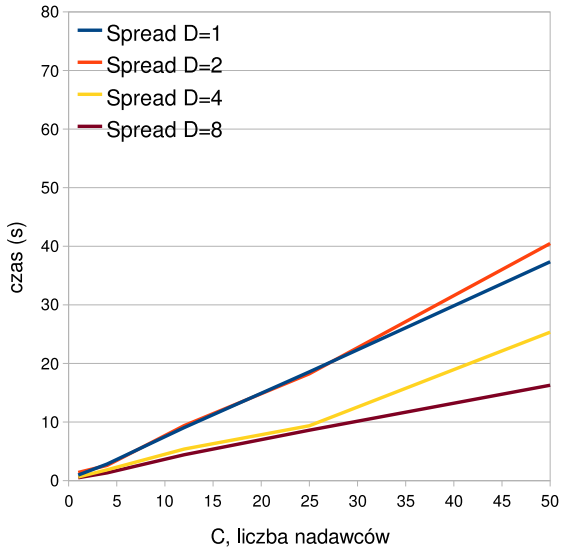
Rysunek 5.5 przedstawia wyniki testu dla $S = 50$ ($\overline{\sigma_{\%}} = 3.95\%$) i $S = 70$ ($\overline{\sigma_{\%}} = 4.81\%$). W przypadku obu porównywanych narzędzi osiągane czasy są większe niż przy wiadomościach 45 bajtowych (por. rysunek 5.4), jednakże przewaga Multiplexera nad Spreadem jest w tym teście istotnie większa. Ponieważ między demonami Multiplexera nie ma żadnych zależności, Multiplexer lepiej zrównolegla swoją pracę i efektywniej wykorzystuje dostępne pasmo. Przy $C = 50$ nadawcach, $S = 70$ odbiorcach i $D = 15$ demonach uruchomienie konfiguracji testowej z Multiplexerem trwało średnio 14.36 s. Ponieważ każdy nadawca wysyłał $T = 5000$ wiadomości po 1024 bajty, osiągnięty został logiczny transfer danych wielkości:

$$C * S * T * 1024 \text{ bajty } / \ 14.36 \text{ s} = 1190.1 \text{ MiB/s}.$$

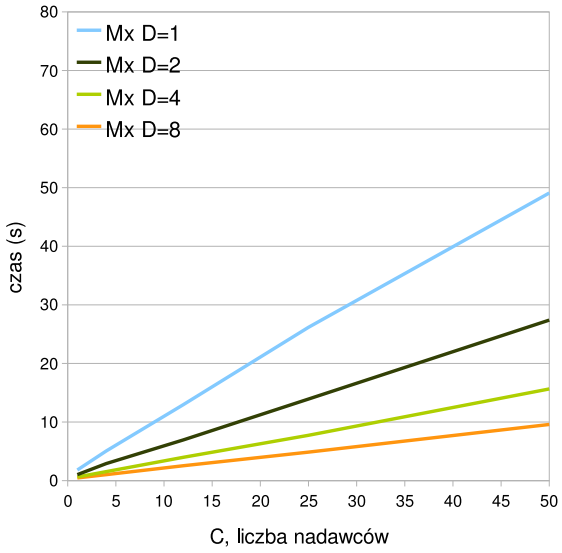
Powyższa równość nie uwzględnia faktu, że każda wiadomość musiała być przesłana do jednego z demonów zanim została rozesłana do odbiorców, a więc każda wiadomość została przesłana $S + 1$ razy. Uwzględniając około 12 bajtowy nagłówek każdej wiadomości otrzymujemy fizyczny transfer pomiędzy poszczególnymi uczestnikami komunikacji wielkości

$$C * (S + 1) * T * 1036 \text{ bajtów } / \ 14.36 \text{ s} \approx 1220 \text{ MiB/s} \approx 9.5 \text{ Gibps},$$

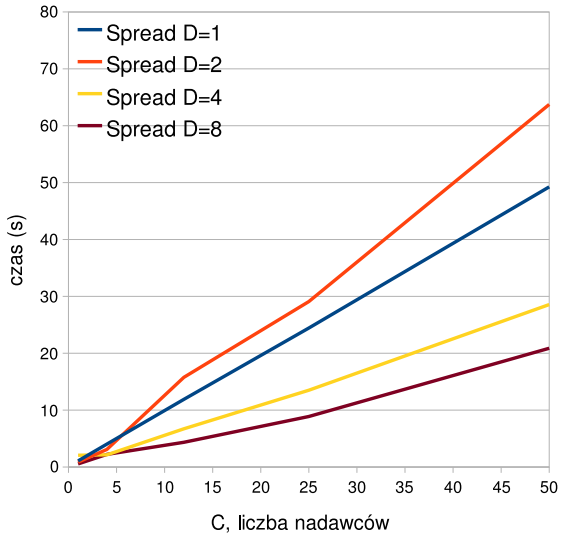
co jest możliwe w gigabitowej sieci lokalnej wyłącznie w przypadku, gdy jest ona oparta na przełącznikach (a nie hubach), przy równoległej i bezkolizyjnej komunikacji pomiędzy poszczególnymi komputerami.



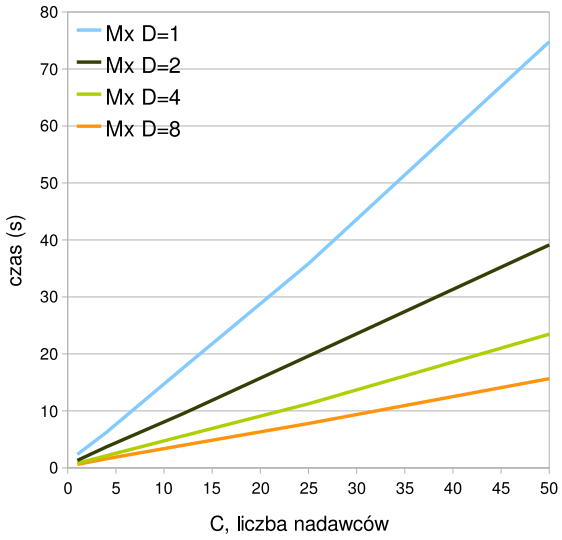

**(a)** Spread, $S = 50$




**(b)** Multiplexer, $S = 50$




**(c)** Spread, $S = 70$




**(d)** Multiplexer, $S = 70$

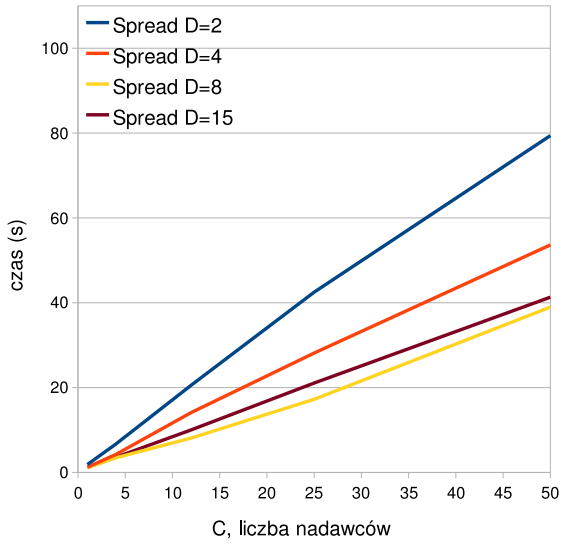
**Rysunek 5.4:** Rozsyłanie powiadomień, 50 i 70 odbiorców

### 5.2.5. Test: symulowana awaria sieci

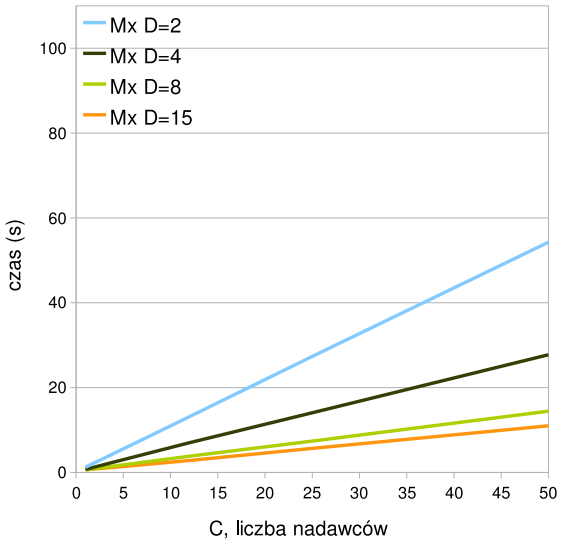
Celem tego testu było sprawdzenie działania Spreada i Multiplexera w sytuacji awarii części węzłów. Odporność na awarie jest kluczowa dla zapewnienia wysokiej dostępności narzędzia do komunikacji (por. sekcja 1.1.1).

#### Scenariusz

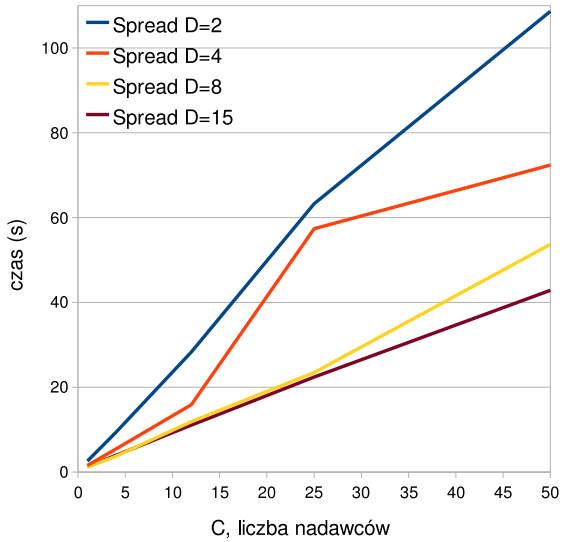
W tym teście $C = 12$ nadawców wysyłało $T = 5000$ powiadomień po 45 bajtów poprzez $D = 10$ demonów do $S = 100$ odbiorców. Testowane były trzy konfiguracje:



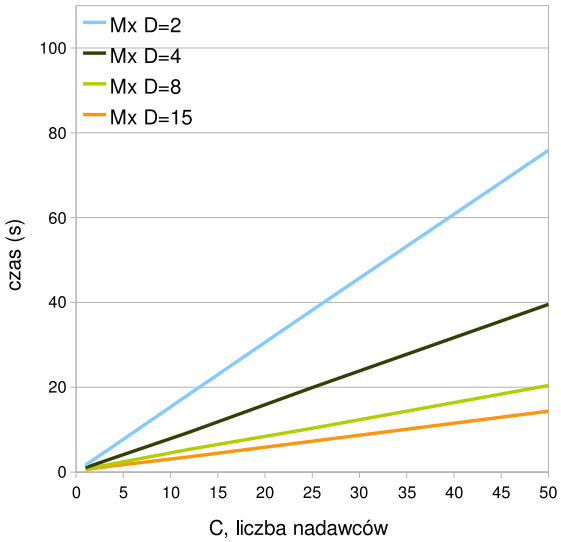

**(a)** Spread, $S = 50$




**(b)** Multiplexer, $S = 50$




**(c)** Spread, $S = 70$




**(d)** Multiplexer, $S = 70$

**Rysunek 5.5:** Rozsyłanie dużych powiadomień, 50 i 70 odbiorców

**Spread** nadawcy wysyłali powiadomienia do wspólnej grupy odbiorców, podobnie jak w testach 5.2.3 i 5.2.4,

**Mx** każde powiadomienie było wysyłane przez jeden demon Multiplexera i dalej rozsyłane do odbiorców, podobnie jak w testach 5.2.3 i 5.2.4,

**Mx Event** każde powiadomienie wysyłane było mechanizmem event, czyli przez wszystkie aktualnie podłączone demony Multiplexera (por. sekcja 4.4.2); odbiorcy usuwali zduplikowane wiadomości na podstawie unikatowych identyfikatorów wiadomości.

Każda konfiguracja testowana była w 10 przypadkach dla $K = \{0, 1, \ldots, 9\}$. W każdej konfiguracji, przy określonym parametrze $K$, test rozpoczynał się uruchomieniem demonów, odbiorców i nadawców. Po pierwszej sekundzie od uruchomienia nadawców $K$ demonów było

wstrzymywanych poprzez sygnał `SIGKILL`, w odstępach 0.1 sekundy. Procesy nadawców kończyły się zaraz po wysłaniu ostatniego powiadomienia. Pozostałe demony były wstrzymywane minutę po zakończeniu procesów nadawców. Wynikiem testu był procent dostarczonych powiadomień oraz czas działania nadawców.

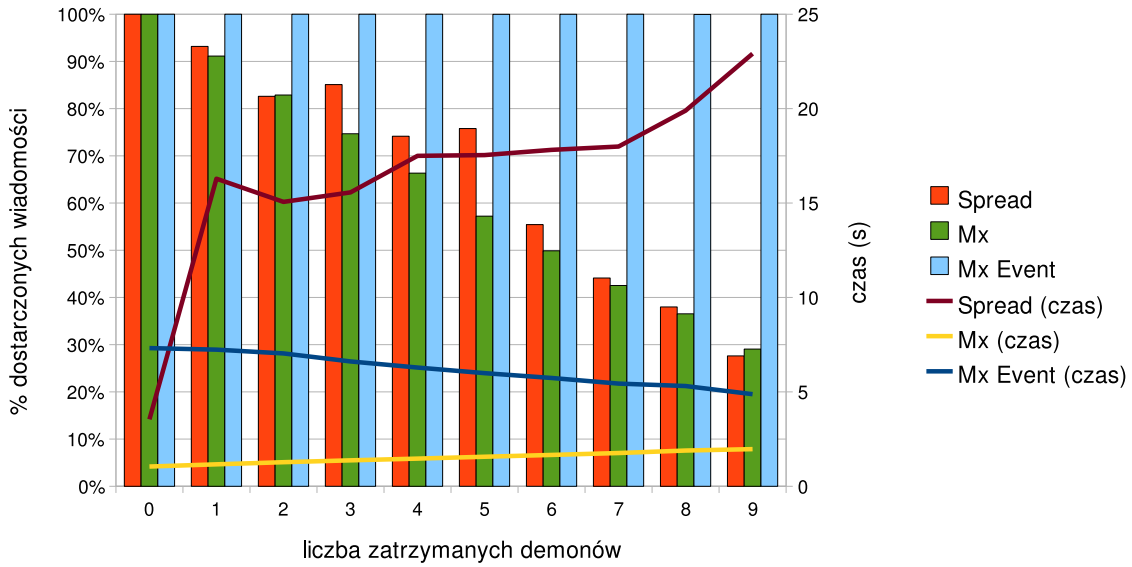
### Wyniki

W następującej tabeli przedstawione zostały skuteczność i czas działania poszczególnych konfiguracji w zależności od liczby wstrzymywanych demonów (parametr $K$). Skutczność („skut.") i procentowe odchylenie standardowe ($\sigma_{\%}$) podane są w procentach, a czas – w sekundach. Zauważmy, że mimo jednakowych warunków przeprowadzania testu, średnie procentowe odchylenie standardowe, $\overline{\sigma_{\%}}$, dla czasu i skuteczności jest w przypadku Spreada bardzo duże. Świadczyć to może o dużej wrażliwości Spreada na czynniki zewnętrzne i jego nieco nieprzewidywalnym zachowaniu.

| $K$ | *Spread* | | | | *Mx* | | | | *Mx Event* | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | skut. | $\sigma_{\%}$ | czas | $\sigma_{\%}$ | skut. | $\sigma_{\%}$ | czas | $\sigma_{\%}$ | skut. | $\sigma_{\%}$ | czas | $\sigma_{\%}$ |
| 0 | 100.0 | 0.00 | 3.54 | 12.8 | 100.0 | 0.00 | 1.05 | 0.84 | 100.0 | 0.00 | 7.32 | 0.80 |
| 1 | 93.2 | 5.57 | 16.29 | 8.32 | 91.1 | 0.09 | 1.16 | 0.21 | 100.0 | 0.00 | 7.24 | 1.34 |
| 2 | 82.6 | 9.10 | 15.06 | 1.47 | 82.9 | 0.22 | 1.27 | 0.59 | 100.0 | 0.00 | 7.03 | 2.25 |
| 3 | 85.1 | 10.2 | 15.56 | 7.09 | 74.7 | 0.97 | 1.37 | 0.52 | 100.0 | 0.00 | 6.61 | 0.91 |
| 4 | 74.2 | 18.6 | 17.50 | 13.8 | 66.3 | 0.74 | 1.47 | 0.43 | 100.0 | 0.00 | 6.29 | 3.05 |
| 5 | 75.8 | 15.9 | 17.54 | 4.45 | 57.2 | 1.05 | 1.56 | 0.13 | 100.0 | 0.04 | 5.99 | 1.01 |
| 6 | 55.4 | 9.62 | 17.82 | 13.9 | 49.9 | 0.51 | 1.66 | 0.43 | 100.0 | 0.02 | 5.74 | 1.81 |
| 7 | 44.1 | 16.4 | 17.99 | 21.7 | 42.5 | 2.01 | 1.76 | 0.27 | 100.0 | 0.00 | 5.44 | 1.54 |
| 8 | 38.0 | 4.34 | 19.89 | 20.4 | 36.5 | 2.51 | 1.89 | 1.34 | 100.0 | 0.03 | 5.31 | 0.91 |
| 9 | 27.6 | 7.18 | 22.91 | 0.24 | 29.1 | 2.13 | 1.97 | 0.20 | 100.0 | 0.00 | 4.88 | 3.26 |
| $\overline{\sigma_{\%}}$ | | 9.69 | | 10.4 | | 1.02 | | 0.50 | | 0.009 | | 1.69 |

Dane z tabeli zostały także przedstawione na rysunku 5.6. Zauważmy, że czas testu przedstawiony na tym rysunku dla konfiguracji *Spread* i *Mx* jest mniejszy, niż można by się spodziewać na podstawie wyników testu 5.2.3 dla podobnych parametrów $S$, $C$, $D$ i $T$, ponieważ w teście 5.2.3 nadawcy *oczekiwali na potwierdzenie* otrzymania wszystkich wiadomości przez odbiorców.

Procent dostarczonych wiadomości w konfiguracji *Mx* maleje liniowo wraz ze wzrostem parametru $K$. Naturalnie każde wstrzymanie demona oznacza zgubienie wiadomości znajdujących się aktualnie w jego pamięci oraz wysłanych do niego, zanim nadawcy stwierdzą jego awarię. Wyniki konfiguracji *Spread* są podobne, choć nieco lepsze, co sugeruje, że demony Spreada buforują mniej informacji w pamięci. Na uwagę zasługuje czas wykonania testu przy $K > 0$ w porównaniu z czasem wykonania przy $K = 0$. Awarie demonów Multiplexera w konfiguracji *Mx* nieznacznie wydłużają wykonanie testu, ponieważ zmniejsza się współbieżność systemu. Natomiast w konfiguracji *Mx Event* czas działania jest tym mniejszy, im większe jest $K$, ponieważ każda wiadomość jest dostarczana wielokrotnie – tyle razy, ile jest aktywnych demonów Multiplexera. W konfiguracji *Spread* po wstrzymaniu pierwszego demona czas wykonania testu gwałtownie rośnie, ponieważ Spread w takiej sytuacji musi się zrekonfigurować, a to wymaga czasu. Wstrzymywanie kolejnych demonów w okresie rekonfiguracji tylko nieznacznie wydłuża czas jej wykonania, więc nie ma już tak dużego wpływu na czas wykonania całego testu.




**Rysunek 5.6:** Skuteczność narzędzi przy awariach węzłów

Zauważmy, że niezależnie od wartości $K$ w konfiguracji *Mx Event* wszystkie wiadomości są poprawnie dostarczane do odbiorców, jednak dzieje się to kosztem nawet siedmiokrotnie dłuższego czasu wysyłania wiadomości. W praktycznych zastosowaniach konieczne jest znalezienie równowagi pomiędzy pewnością dostarczenia, a obciążeniem sieci, jakie możemy generować. Wysyłanie wiadomości przez 3 niezależne demony Multiplexera gwarantuje dostarczenie wiadomości, o ile odbiorcy odczytują wiadomości z dostateczną prędkością (tj. nie dochodzi do przepełnienia kolejek wiadomości oczekujących na dostarczenie) i nie ma rozległych awarii sieci.

## 5.3. Wnioski

Wyniki testów opisane w tym rozdziale pokazują, że Multiplexer sprawdza się dużo lepiej niż Spread w typowych zastosowaniach biznesowych, a więc lepiej spełnia wymagania systemu rozproszonego portalu Azouk oraz podobnych systemów. Spread jest niezastąpiony, gdy od narzędzia do komunikacji wymagamy ustalania liniowego porządku wiadomości lub automatycznego rozsyłania powiadomień o nowych węzłach podłączonych do systemu. Implementacja Spreada jest wydajniejsza przy małej liczbie wymienianych pakietów. Multiplexer dobrze sprawdza się przy większej liczbie wymienianych pakietów oraz jako pośrednik w dostępie do usług, czyli przy zadaniach, dla których został zaprojektowany.

# Rozdział 6

# Podsumowanie

Jak wykazałem w rozdziale 4, Multiplexer spełnia stawiane mu wymagania funkcjonalne, opisane w rozdziale 2. Multiplexer posiada takie pożądane cechy narzędzi do komunikacji (por. sekcja 1.1) jak: model publikacji-subskrypcji, wysoka dostępność, duża przepustowość, małe opóźnienia, pewność dostarczenia, łatwe zbieranie informacji diagnostycznych, rozszerzalność i przenośność protokołu. Przeprowadzone testy pokazały, że Multiplexer radzi sobie lepiej niż konkurencyjne narzędzia w typowych dla niego zastosowaniach. Tak więc wszystkie założenia niniejszej pracy zostały spełnione.

Dzięki odpowiedniej architekturze demony Multiplexera są niezależne, co zapewnia właściwie nieograniczoną skalowalność całego systemu oraz odporność na awarie. Elastyczny i rozszerzalny protokół komunikacyjny umożliwia dodawanie nowych funkcjonalności z zachowaniem kompatybilności wstecz. Dzięki zastosowaniu biblioteki Google Protocol Buffers, protokół jest nie tylko elastyczny, ale również przenośny i szybki w kodowaniu i dekodowaniu.

Multiplexer jest stosowany zarówno w rozwojowej, jak i produkcyjnej wersji portalu Azouk. Nie tylko spełnia wymagania tego portalu, ale też rzeczywiście ułatwia budowanie rozproszonego systemu usług i zarządzanie nim.

Poniżej prezentuję możliwe rozszerzenia funkcjonalności Multiplexera, które nie wchodziły w skład stawianych mu wymagań, a mogą ułatwić korzystanie z niego i umożliwić szerszy zakres jego stosowania.

## 6.1. Możliwe rozszerzenia

### 6.1.1. Trwałość wiadomości

W niektórych zastosowaniach pożądana jest trwałość wiadomości (ang. *message persistence*), tj. by wysłana wiadomość była skutecznie dostarczona, bez względu na czas doręczenia. Trwała wiadomość powinna być dostarczona także w przypadku, gdy podczas wysyłania wiadomości klient nie jest podłączony do żadnego serwera Multiplexera, a więc trwałość wiadomości musi być implementowana w bibliotece klienckiej.

Opcjonalnie trwała wiadomość powinna móc być dostarczona także wówczas, gdy odbiorca i nadawca nigdy nie są równocześnie podłączeni do sieci. Taka funkcjonalność wymaga implementacji trwałych kolejek wiadomości bezpośrednio w serwerach Multiplexera lub w dedykowanych usługach wspierających.

### 6.1.2. Priorytety

Naturalnym rozszerzeniem funkcjonalności przekazywania wiadomości są priorytety. Po pierwsze, priorytety wiadomości mogą określać kolejność, w jakiej wiadomości są dostarczane do odbiorców. Po drugie, mogą określać, czy wiadomość może być odrzucana przez serwer Multiplexera w sytuacji przepełnienia kolejek wiadomości oczekujących dla danego odbiorcy.

### 6.1.3. Blokowanie szybkich nadawców

W systemie, w którym istnieje szybki nadawca i powolny odbiorca, niektóre wiadomości są konsekwentnie gubione, ponieważ kolejka wiadomości oczekujących dla odbiorcy jest permanentnie przepełniona. Rozwiązaniem tego problemu mogłoby być blokowanie nadawcy przez serwer zamiast odrzucania jego wiadomości w sytuacji przepełnienia kolejki odbiorcy. Implementując blokowanie szybkiego nadawcy, należałoby zwrócić szczególną uwagę na problem potencjalnego zakleszczenia i właściwą obsługę priorytetów wiadomości.

### 6.1.4. Przedawnianie wiadomości

W sytuacji, w której odbiorca wiadomości zajęty jest jednym długim zadaniem i nie przetwarza przychodzących wiadomości na bieżąco, w jego kolejce wiadomości do obsłużenia może nagromadzić się wiele zadań, także takich, na których wykonanie nikt już nie czeka (zadania przedawnione). Możliwe są dwa sposoby radzenia sobie z takimi sytuacjami. Pierwszym z nich jest oznaczenie czasu ważności wiadomości. Takie rozwiązanie wymaga, aby zegary komunikujących się aplikacji były zsynchronizowane. Drugim sposobem jest przedawnianie wiadomości – aplikacja, która rezygnuje z oczekiwania na wykonanie konkretnego zadania, mogłaby wysyłać wiadomość anulującą to zadanie. To rozwiązanie wymaga, by „wiadomości anulujące" przetwarzane były przed zadaniami (miały wyższy priorytet).

### 6.1.5. Bezpieczeństwo

Ponieważ Multiplexer jest przeznaczony do zastosowań intranetowych, dostarczany przez niego poziom bezpieczeństwa jest bardzo niski. Opcjonalna autoryzacja połączeń, dokonywana przez serwer, używa hasła przesyłanego otwartym tekstem i dobrze nadaje się do zabezpieczania systemu przed pomyłkowym użyciem – np. system produkcyjny można łatwo oddzielić od testowego używając różnych haseł – ale nie do zabezpieczania przed włamaniem.

Obecnie bezpieczne używanie Multiplexera na otwartych, internetowych połączeniach wymaga stosowania wirtualnych sieci prywatnych (ang. *Virtual Private Network*). W przyszłości możliwe jest dodanie wsparcia dla połączeń szyfrowanych, na przykład korzystających z obsługi SSL dostarczanej przez bibliotekę Asio.

### 6.1.6. Przenośność

Protokół Multiplexera oparty jest na bibliotece Google Protocol Buffers, zapewniającej przenośność na różne platformy. Oficjalna dystrybucja Protocol Buffers umożliwia implementację protokołu w C++, Pythonie i w Javie. Dzięki Javie i Pythonowi możliwe jest stosowanie protokołu na różnych architekturach sprzętowych i systemach operacyjnych. Dodatkowo, jak wspomniałem w sekcji 4.2.2, istnieją nieoficjalne implementacje Google Protocol Buffers dla platformy .NET i takich języków jak C, Erlang, Haskell, Perl, co umożliwia implementację protokołu Multiplexera również w tych językach.

W ramach projektu sponsorowanego przez Wydział Fizyki Uniwersytetu Warszawskiego została stworzona biblioteka kliencka Multiplexera dla Javy, obsługująca asynchronicznie operacje wejścia-wyjścia z wykorzystaniem Java NIO [16], za pośrednictwem biblioteki JBoss Netty [17].

### 6.1.7. Integracja

Poza implementacjami protokołu Multiplexera w różnych językach programowania, dużym ułatwieniem dla szerokiej akceptacji Multiplexera byłoby wsparcie dla już istniejących protokołów i interfejsów. Szczególnie należałoby rozważyć możliwość dodania obsługi protokołu AMQP (por. sekcja 3.2.3) w serwerze Multiplexera oraz stworzenie biblioteki klienckiej dla Javy zgodnej ze standardem Java Message Service (por. sekcja 3.2.2).

# Dodatek A

# Zawartość płyty CD

Płyta CD dołączona do niniejszej pracy zawiera następujące pliki i katalogi:

**pfindeisen-praca-magisterska.pdf** – wersja pracy w formacie PDF,

**zrodla** – źródła serwera Multiplexera, biblioteki klienckiej oraz programów pomocniczych do przesyłania logów,

**testy** – katalog z wersjami użytymi do testowania wraz z programami testującymi,

**testy/spread** – wersja Spreada użyta do testów,

**testy/spread/examples/test-*** – skrypty testujące dla Spreada,

**testy/multiplexer** – wersja Multiplexera użyta do testów (różnice między tą wersją a zawartością katalogu `zrodla` zostały wprowadzone jedynie po to, by ułatwić testowanie poprzez szybsze wykrywanie sytuacji błędnych),

**testy/multiplexer/src/azouk/test-*** – skrypty testujące dla Multiplexera,

**wyniki-testow** – katalog z wynikami testów w formacie używanym przez skrypty testujące oraz programami do ich konwersji do formatu CSV,

**wykresy** – wykresy użyte w pracy w formacie EPS,

**README.txt** – plik z opisem zawartości płyty.

# Dodatek B

# Formaty danych

## B.1. MultiplexerMessage

Oto definicja klasy `MultiplexerMessage` (por. sekcja 4.3.2).

```
message MultiplexerMessage {
    // packet ID; generally required but see
    // http://code.google.com/apis/protocolbuffers/docs/proto.html#simple
    optional uint64 id = 1;

    // ID of the sender; generally required
    optional uint64 from = 2;

    // send a message directly to a peer with ID 'to';
    // overrides all rules, also 'override_rrules'
    optional uint64 to = 3;

    // report error delivery when routing via MultiplexerMessage.to
    optional bool report_delivery_error = 21 [default = false];
    optional bool include_original_packet_in_report = 22 [default = false];

    // packet type
    required uint32 type = 4;

    // the actual message we want to send
    optional bytes message = 5;
    optional Compression.Values compression = 24 [default = NO_COMPRESSION];

    // when the packet was sent
    optional uint32 timestamp = 6;

    // packet ID to which we reply
    // this may be change to repeated field in the future
    optional uint64 references = 7;

    // work flow ID
    optional bytes workflow = 8;

    // support for generally overriding the default routing rules
    repeated MultiplexerMessageDescription.RoutingRule override_rrules = 20;

    // request disabling logging to file or stream
    optional LoggingMethod.Values logging_method = 23 [default = BOTH];
}
```

Dalej znajdują się definicje klas używanych przez klasę `MultiplexerMessage`. Zauważmy, że format `MultiplexerMessageDescription` jest używany także przy opisie pliku konfiguracyjnego.

```
message MultiplexerMessageDescription {

    message RoutingRule {
        enum Whom {
            ALL = 1; // forward message to all peers of given type
            ANY = 2; // forward message to one peer of given type
        }

        // peer name, e.g. "SEARCH"; required in configuration files
        optional string peer = 20;

        // peer type, e.g. types::SEARCH; required in binary format
        optional uint32 peer_type = 1;

        // how many peers forward to? defaults to ANY
        optional Whom whom = 2 [default = ANY];

        // is a peer presence required?
        optional bool delivery_error_is_error = 3 [default = true];

        // inform the sender about delivery error?
        optional bool report_delivery_error = 4 [default = true];
        optional bool include_original_packet_in_report = 5 [default = false];
    }

    required uint32 type = 1;
    required string name = 2;
    optional string comment = 3;

    // defines routing rules for given packet
    // the first entry here will be used in processing BackendForPackerSearch
    repeated RoutingRule to = 4;
}

message Compression {
    enum Values {
        NO_COMPRESSION = 0;
        GZIP = 1;
    };
}

message LoggingMethod {
    // this message is only a namespace for the following enum
    enum Values {
        // this are bit-flags
        CONSOLE = 1;
        FILE = 2;
        BOTH = 3;
    }
}
```

## B.2. Plik konfiguracyjny

Format pliku konfiguracyjnego Multiplexera jest tekstowym formatem serializacji klasy `MultiplexerRules`.

```
message MultiplexerRules {
    repeated MultiplexerMessageDescription type = 1;
    repeated MultiplexerPeerDescription peer = 2;
}

message MultiplexerPeerDescription {

    // ID of this type
    required uint32 type = 1;

    // name of this type for use in text-based config files
    required string name = 2;

    optional string comment = 3;

    // how many pending messages this peer can have (e.g. how big
    // a queue can be)
    optional uint32 queue_size = 4 [default = 1024];

    // some clients are fully passive, so e.g. we should not require heartbits
    // from them
    optional bool is_passive = 5 [default = false];
}
```

Przykładowy plik konfiguracyjny Multiplexera wygląda następująco:

```
# peer definitions
#
#   PEERS 1 - 99 reserverd for Multiplexer and special types
peer {
    type: 1
    name: "MULTIPLEXER"
    comment: "peer type representing normal multiplexer instance."
}

peer {
    type: 2
    name: "ALL_TYPES"
    comment: "special peer type that causes a message to be sent to all types"
}

peer {
    type: 99
    name: "MAX_MULTIPLEXER_SPECIAL_PEER_TYPE"
    comment: "this only defines a constant"
}

#   PEERS 100-999 are plain peers
peer {
    type: 102
    name: "WEBSITE"
    is_passive: true
    comment: "example peer type with is_passive definition"
}

peer {
    type: 108
    name: "LOG_STREAMER"
    is_passive: true
}
```

```
peer {
    type: 109
    name: "LOG_COLLECTOR"
}

peer {
    type: 110
    name: "EVENTS_COLLECTOR"
}

...

# packages and routing rules definitions
#
#   PACKAGES 1 - 99 reserved for Multiplexer meta packages
type {
    type: 1
    name: "PING"
    comment: "I'm alive packet; it never carries any significant message."
}

type {
    type: 2
    name: "CONNECTION_WELCOME"
    comment: "message interchange by peers just after connecting to each other"
}

type {
    type: 3
    name: "BACKEND_FOR_PACKET_SEARCH"
    comment: "message used by MX client in query() for finding working backends"
}

type {
    type: 4
    name: "HEARTBIT"
    comment: "packet sent periodically by every peer on every channel"
}

type {
    type: 5
    name: "DELIVERY_ERROR"
    comment: "packet could not be delivered to one or more recipients"
}

type {
    type: 99
    name: "MAX_MULTIPLEXER_META_PACKET"
    comment: "this only defines a constant"
}

# PACKAGES 100 - 999 are normal packages

type {
    type: 113
    name: "REQUEST_RECEIVED"
    comment: "sent by the backend immediately after receiving a request"
}

type {
    type: 114
    name: "BACKEND_ERROR"
    comment: "sent by the backend when request handling function finishes with error"
}

type {
    type: 115
    name: "LOGS_STREAM"
```

```
    comment: "payload is LogEntriesMessage"
    to {
        peer: "LOG_COLLECTOR"
        whom: ALL
    }
}

type {
    type: 117
    name: "SEARCH_COLLECTED_LOGS_REQUEST"
    # SearchCollectedLogs are Protocol Buffer class defined elsewhere
    comment: "payload is SearchCollectedLogs"
    to {
        peer: "LOG_COLLECTOR"
        whom: ANY
    }
}

type {
    type: 118
    # LogEntriesMessage is Protocol Buffer class defined elsewhere
    comment: "logs are returned in LogEntriesMessage"
    name: "SEARCH_COLLECTED_LOGS_RESPONSE"
}

type {
    type: 126
    name: "REPLAY_EVENTS_REQUEST"
    comment: "this is a no-response request"
    to {
        peer: "EVENTS_COLLECTOR"
        whom: ANY
    }
}
```

# Bibliografia


[1] Advanced Message Queuing Protocol, specyfikacja protokołu w wersji 0-10, http://jira.amqp.org/confluence/download/attachments/720900/amqp.0-10.pdf 2008

[2] Julie D. Allen, Joseph Becker, Unicode Consortium, *The Unicode standard 5.0*, Addison-Wesley, 2006

[3] Apache Qpid, strona projektu, http://cwiki.apache.org/qpid/

[4] Asio, strona projektu, http://think-async.com/

[5] Azouk, portal, http:///www.azouk.com/

[6] Boost, strona projektu, http://www.boost.org/

[7] Don Box, *Essential COM*, Addison-Wesley, 1998

[8] Michael Burrows, *The Chubby Lock Service for Loosely-Coupled Distributed Systems*, OSDI'06: Seventh Symposium on Operating System Design and Implementation

[9] George F. Coulouris, Jean Dollimore, Tim Kindberg, *Systemy rozproszone, podstawy i projektowanie*, WNT 1998

[10] Delicious, portal, http://delicious.com/

[11] Django, strona projektu, http://www.djangoproject.com

[12] Google Protocol Buffers, strona projektu, http://code.google.com/apis/protocolbuffers/

[13] Juan-Mariano de Goyeneche, *Multicast over TCP/IP HOWTO, Multicast Transport Protocols*, http://tldp.org/HOWTO/Multicast-HOWTO-9.html, 1998

[14] William Grosso, *Java RMI: Designing & Building Distributed Applications*, O'Reilly, 2002

[15] IBM WebSphere MQ, strona produktu, www.ibm.com/webspheremq

[16] Java NIO, dokumentacja pakietu `java.nio`, http://java.sun.com/j2se/1.5.0/docs/guide/nio/

[17] JBoss Netty, strona projektu, http://jboss.org/netty/

[18] Louise E. Moser, Yair Amir, P. M. Melliar-Smith, Deborah A. Agarwal. *Extended Virtual Synchrony*, Proceedings of the 14th International Conference on Distributed Computing Systems, 1994.

[19] OpenAMQP, strona projektu, http://www.openamq.org/

[20] py-amqplib, strona projektu, http://barryp.org/software/py-amqplib/

[21] Dirk Slama, Jason Garbis, Perry Russell, *Enterprise CORBA*, Prentice Hall, 1999

[22] Mark Slee, Aditya Agarwal, Marc Kwiatkowski, *Thrift: Scalable Cross-Language Services Implementation*, http://developers.facebook.com/thrift/thrift-20070401.pdf, 2007

[23] Spread, strona projektu, http://www.spread.org/

[24] Jonathan R. Stanton, *A Users Guide to Spread*, http://www.spread.org/docs/guide/users_guide.pdf, 2002

[25] StumbleUpon, portal, http://www.stumbleupon.com/

[26] Twisted, strona projektu, http://twistedmatrix.com/trac/

[27] txAMQP, strona projektu, https://launchpad.net/txamqp

[28] W3C, *SOAP Version 1.2*, http://www.w3.org/TR/soap12-part1/

[29] Wikipedia, *Advanced Message Queuing Protocol*, http://en.wikipedia.org/wiki/Advanced_Message_Queuing_Protocol

[30] Wikipedia, *CORBA, Problems and criticism*, http://en.wikipedia.org/wiki/CORBA#Problems_and_criticism

[31] Wikipedia, *IP Multicast*, http://en.wikipedia.org/wiki/IP_Multicast

[32] Wikipedia, *XML-RPC*, http://en.wikipedia.org/wiki/XML-RPC

[33] Dave Winer, *XML-RPC Specification*, http://www.xmlrpc.com/spec, 1999