Uniwersytet Warszawski
Wydział Matematyki, Informatyki i Mechaniki

Łukasz Heldt
Nr albumu: 181262

Klaster w serwerze WWW Jakarta-Tomcat

Praca magisterska
na kierunku INFORMATYKA

Praca wykonana pod kierunkiem
dr Janiny Mincer-Daszkiewicz
Instytut Informatyki

Wrzesień 2004

Pracę przedkładam do oceny

Data: Podpis autora pracy:

Praca jest gotowa do oceny przez recenzenta

Data: Podpis kierującego pracą:

Streszczenie

W ramach niniejszej pracy zaprojektowano i zaimplementowano nowy moduł do obsługi klastra dla serwera WWW Jakarta-Tomcat. W części pisemnej pracy przedstawiono definicję klastra, podstawowe funkcje, które musi on spełniać w kontekście serwerów WWW oraz problemy, które trzeba rozwiązać w czasie jego implementacji. Na część praktyczną składa się realizacja przedstawionego projektu oraz wykonanie testów wydajności stworzonego modułu, ukazujących jego wady i zalety w praktycznych zastosowaniach.

Słowa kluczowe

Klaster, Java, komunikacja sieciowa, aplikacja internetowa, serwer WWW, HTTP, HTTPS

Klasyfikacja tematyczna

C. Computer Systems Organization
C.2 COMPUTER-COMMUNICATION NETWORKS
C.2.4 Distributed Systems

Spis treści

Spis treści 5
1 Wstęp 7
2 Klastry 8
2.1 Definicja 8
2.1.1 Klaster o wysokiej wydajności 8
2.1.2 Klaster równoważący obciążenie 8
2.1.3 Klaster odporny na awarie 9
2.2 Rozwój systemów klastrowych 9
2.3 Klastry w serwerach WWW 10
2.3.1 Występujące problemy 10
2.3.2 Stosowane rozwiązania 10
3 Jakarta-Tomcat 5.0 13
3.1 Rozwój serwera Jakarta-Tomcat 13
3.2 Główne założenia koncepcyjne 13
3.3 Opis implementacji 14
3.3.1 Hierarchia obiektów 14
3.3.2 Strumień przetwarzania żądań 16
3.3.3 Wyzwalacze 16
3.4 Implementacja klastra w serwerze Tomcat 5.0 17
3.4.1 Klaster dla wersji 4.0 17
3.4.2 Architektura klastra w wersji 5.0 17
3.4.3 Implementacja klastra w wersji 5.0 18
3.4.4 Konfiguracja klastra w wersji 5.0 21
3.4.5 Zalety rozwiązania 22
3.4.6 Wady rozwiązania 23
4 Nowy klaster dla serwera Jakarta-Tomcat 25
4.1 Architektura i założenia koncepcyjne 25
4.1.1 Koncepcja centralnego sterowania 25
4.1.2 Podział na warstwy 26
4.1.3 Wersjonowanie sesji 27
4.1.4 Zintegrowany mechanizm równoważenia obciążenia 27
4.1.5 Bezpieczeństwo 28
4.2 Działanie systemu 28
4.2.1 Dołączanie nowego węzła 29
4.2.2 Awaria połączenia 31
4.2.3 Awaria węzła 32
4.2.4 Awaria serwera zarządcy 33
4.3 Implementacja 34
4.3.1 Warstwa transportująca 35
4.3.2 Komunikacja w klastrze 39
4.3.3 Mechanizm kolejki zadań zależnych 41
4.3.4 Buforowanie 45
4.3.5 Algorytm przetwarzania żądania 47
4.3.6 Serwer zarządca 51
4.3.7 Serwer ruchu 53
4.4 Konfiguracja klastra 54
4.4.1 Koncepcja dwóch sieci 56

4.4.2 Możliwe konfiguracje przy wykorzystaniu HTTPS 56
5 Testy wydajności 58
5.1 Test poprawności 59
5.2 Test wydajności 61
5.3 Test HTTPS 62
6 Możliwe rozszerzenia 64
6.1 Algorytm elekcji 64
6.2 Podział klastra na podgrupy 64
6.3 Zmniejszanie ilości przesyłanych informacji 65
6.4 Implementacja interfejsu użytkownika do zarządzania klastrem 65
6.5 Rozbudowa serwera ruchu 66
7 Podsumowanie 67
Dodatek A Opis załączników 68
Bibliografia 69

1 Wstęp

W dzisiejszych czasach usługi informatyczne zdominowały zarówno świat biznesu, jak i prywatne życie zwykłych ludzi. Internet spopularyzował systemy komputerowe do tego stopnia, że zaczynają one wypierać tradycyjne metody przetwarzania informacji ze wszystkich dziedzin życia. W świecie WWW można skorzystać ze słownika, obejrzeć mapy, szybko i komfortowo wyszukać dowolne interesujące nas dane.

Jak łatwo się domyśleć wraz z popularnością systemów informatycznych wzrosła liczba ich potencjalnych użytkowników, co bezpośrednio wiąże się z potrzebą zwiększenia mocy obliczeniowej. Firmy prześcigają się w ofertach dużych, wieloprocesorowych maszyn, które umożliwiłyby obsługę tysięcy czy nawet milionów użytkowników na sekundę. Niestety koszt takiej maszyny bardzo często przekracza budżet małych czy średnich firm, które mają dobre pomysły, ale brakuje im odpowiednich funduszy, aby je zrealizować. W takiej sytuacji idealnym rozwiązaniem może okazać się połączenie wielu zwykłych, niedrogich komputerów, które wspólnie umożliwią obsługę dużej liczby klientów. Aby taka architektura mogła rozwiązać problem zwiększonego zapotrzebowania na moc obliczeniową w sposób niewidoczny dla końcowego użytkownika, niezbędne jest odpowiednie oprogramowanie.

W swojej pracy przedstawię pomysł stworzenia modułu wspierającego rozproszone przetwarzanie danych (klaster) dla bardzo popularnego serwera WWW – Jakarta-Tomcat. W pierwszej części zostanie przedstawiona koncepcja klastra (por. rozdz. 2) oraz opisany sposób działania Tomcata (por. rozdz. 3). W drugiej części pracy przedstawię zrealizowany przeze mnie moduł klastra (por. rozdz. 4) wraz z testami jego wydajności (por. rozdz. 5). Możliwe rozszerzenia opisuję w rozdz. 6.

W pracy prezentuję pozytywne strony wykorzystania modelu replikowania „każdy do każdego”. Wykazuję, że dobrze zaimplementowany moduł może okazać się bezpieczniejszy i wydajniejszy od modelu „serwer macierzysty, serwer kopii”. Oczywiście implementacja jest znacznie bardziej skomplikowana (pojawia się problem synchronizowania dostępu do sesji) – niemniej jednak zysk jaki można osiągnąć stosując ten model w przypadku protokołu HTTPS rekompensuje trudy implementacji modułu.

2 Klastry

2.1 Definicja

W literaturze fachowej pojawia się następująca definicja klastra:

> *Klaster to zbiór niezależnych komputerów (węzłów), połączonych siecią komunikacyjną, które z punktu widzenia użytkownika sprawiają wrażenie pojedynczego systemu (komputera) [2].*

Głównym zadaniem jakie stawia się klastrom jest zwiększenie szybkości, niezawodności i dostępności systemów informatycznych. Rozdzielając obciążenie jednej maszyny na całą farmę maszyn otrzymuje się zrównoleglenie obliczeń, niewidoczne dla końcowego użytkownika. W systemach, które obsługują tysiące lub nawet miliony użytkowników na sekundę rozdzielenie obciążenia na wiele fizycznych maszyn może okazać się jedynym możliwym rozwiązaniem problemu wydajności i zapewnienia ciągłej pracy systemu.

Bardzo ważną cechą, która często powoduje, że administratorzy dużych systemów decydują się na instalację klastra jest jego odporność na awarie pojedynczej maszyny. Jeżeli klaster zapewnia mechanizm niewidocznej dla użytkownika obsługi awarii węzła (ang. *fail-over*), to zastosowanie architektury korzystającej z wielu maszyn fizycznych może być jedynym sposobem uzyskania niezawodności systemu informatycznego. W przypadku awarii węzła klaster zapewnia przeniesienie kontekstu wykonania na inny serwer, a końcowy użytkownik może się nie zorientować, że nastąpiła awaria, gdyż zachowa swoje dane. Ponadto nie mniej ważną zaletą jest skalowalność rozwiązań wykorzystujących klaster. W momencie, gdy administrator zaczyna zauważać spadek mocy obliczeniowej systemu może podłączyć kolejną maszynę, zwiększając przepustowość klastra. Z drugiej strony jeżeli okaże się, że system nie wykorzystuje w pełni zasobów sprzętowych, to część maszyn można odłączyć od klastra, wykorzystując je w innych systemach, na daną chwilę bardziej wymagających.

W szczególności wyróżnia się trzy rodzaje klastrów:

1. Klaster o wysokiej wydajności (ang. *high performance cluster*),
2. Klaster równoważący obciążenie (ang. *load balancing cluster*),
3. Klaster odporny na awarie (ang. *fail over cluster*).

W każdym z trzech rodzajów klastrów kładzie się nacisk na inne aspekty związane z systemami informatycznymi lub sprzętem. W kolejnych podrozdziałach umieszczono krótką charakterystykę każdego z wymienionych rodzajów.

2.1.1 Klaster o wysokiej wydajności

Zadaniem klastra o wysokiej wydajności jest maksymalizowanie mocy obliczeniowej, którą można uzyskać tworząc farmę maszyn. Przy implementacji główny nacisk kładzie się na wydajność systemu (co często wiąże się również z drugim rodzajem klastra – czyli klastrem równoważącym obciążenie).

2.1.2 Klaster równoważący obciążenie

Klaster równoważący obciążenie jest instalowany w celu zapewnienia równego podziału zadań na wszystkie maszyny w klastrze. Tego typu klaster instaluje się w systemach, w których bardzo istotny jest czas reakcji na żądanie klienta. Klaster

minimalizuje wariancję czasu oczekiwania na odpowiedź od systemu. Jest odpowiedzialny za wykorzystywanie wszystkich swoich zasobów sprzętowych w równym stopniu, eliminując sytuacje, gdzie jedno żądanie od klienta trafia do całkowicie niewykorzystywanego węzła i jego obsługa trwa kilka sekund, a inne trafia do mocno obciążonego węzła i obsługa trwa kilkadziesiąt sekund. Najbardziej wyspecjalizowane systemy podczas wyboru węzła do obsługi żądania biorą pod uwagę statystyki obciążenia maszyny (zużycie procesora, dysków, pamięci) z ostatnich kilku sekund i na tej podstawie określają jego potencjalną moc przerobową. Niemniej jednak mniej wyrafinowane algorytmy w przypadku wielu systemów są wystarczająco skuteczne. Przykładem jest najczęściej stosowany algorytm karuzelowy (ang. *Round Robin*), który każde kolejne żądanie wysyła do następnego węzła, w pewnym z góry ustalonym cyklu.

2.1.3 Klaster odporny na awarie

Zadaniem klastra odpornego na awarie jest zapewnienie niezawodności systemu, nawet w przypadku fizycznych bądź programowych awarii. Klaster w zależności od stopnia zaawansowania może obsługiwać odpowiednio duży procent awarii węzłów bez utraty danych – oczywiście żądania przetwarzane w momencie awarii mogą część danych utracić. Odporność na awarie węzłów uzyskuje się w dwojaki sposób: na poziomie sprzętu oraz programowo.

Każdą niezbędną z punktu widzenia funkcjonowania systemu maszynę duplikuje się i w przypadku awarii duplikat przejmuje rolę podstawowej maszyny. Niemniej jednak takie rozwiązanie nie gwarantuje nam, że dane użytkowników trzymane w pamięci operacyjnej maszyny nie zostaną utracone podczas awarii. Dlatego w systemach przechowujących dane związane z sesją użytkownika stosuje się programowe wsparcie dla mechanizmów automatycznego wykrywania i obsługi awarii. Najczęściej spotykane rozwiązania to zapis danych na nośniku fizycznym (na przykład w bazie danych) lub replikacja danych w obrębie węzłów klastra. Oba rozwiązania mają swoje wady i zalety. Zapis danych w bazie umożliwia ich odzyskanie nawet po awarii całego klastra, ale powoduje duży narzut przy obsłudze żądań klientów. Replikacja uniemożliwia odzyskanie danych przy awarii całego klastra, ale daje duże możliwości równoważenia obciążenia – kontekst danego użytkownika nie musi być na stałe powiązany z węzłem, na którym zaczął się wykonywać jego program.

2.2 Rozwój systemów klastrowych

Pierwsze udane próby stworzenia klastra sięgają 1968 roku. Grupa naukowców na Uniwersytecie Illinois w ramach projektu ILLIAC IV [2] stworzyła klaster umożliwiający połączenie setek, a nawet tysięcy węzłów w jeden superkomputer. W czasie realizacji projektu pojawiło się wiele problemów, jednak mimo to idea przetrwała. W późniejszych latach programiści odeszli od koncepcji równoległego przetwarzania, co wiązało się z przekonaniem, że skoro w ciągu dziewięciu miesięcy szybkość mikroprocesora podwaja się, to nie ma potrzeby pisania skomplikowanego oprogramowania wspierającego klastry, w celu zwiększenia mocy obliczeniowej. Prędzej czy później technologia dogoni wymagania programów.

Koncepcja klastrów obroniła się jednak przed pędzącym rozwojem technologii. W ciągu ostatnich kilkunastu lat firmy zaczęły bardzo żywo interesować się tworzeniem oprogramowania (szczególnie dużych serwisów i baz danych) w środowisku rozproszonym, złożonym z wielu fizycznie niezależnych maszyn.

Okazuje się, że w dzisiejszych czasach ważniejsza staje się gwarancja niezawodności oraz dostępności (ang. *high availability*) systemu od jego wydajności. Wydajność można uzyskać kupując wieloprocesorową maszynę, jednak nikt nie zagwarantuje, że taka maszyna będzie działała non-stop przez wiele lat. Instalując klaster firma ma pewność, że nawet w przypadku awarii jednego z węzłów system wciąż będzie dostępny.

Innym aspektem powodującym wzmożone zainteresowanie klastrami jest ich skalowalność. Często liczba użytkowników końcowych systemu jest niemożliwa do oszacowania (szczególnie, jeżeli system jest dostępny w Internecie jako serwer WWW). Bez tej wiedzy projektanci systemu nie są w stanie z góry określić niezbędną moc obliczeniową, która pozwoli użytkownikom systemu na wygodną pracę. Skalowalność i niezawodność klastrów powoduje, że stają się one bardzo atrakcyjne dla twórców takich systemów.

2.3 Klastry w serwerach WWW

Większość dużych systemów informatycznych tworzonych w dzisiejszych czasach w jakimś stopniu korzysta z serwerów WWW. Wynika to z ogromnej rzeszy potencjalnych użytkowników systemu znajdującej się w Internecie, a także z jasno wyznaczonych standardów, które takie serwery spełniają. Pisząc interfejs użytkownika jako strony HTML mamy pewność, że klient będzie w stanie korzystać z serwisu na dowolnej platformie, bez potrzeby doinstalowywania lokalnie dodatkowych komponentów.

Popularność protokołu HTTP spowodowała masowe wykorzystywanie go nawet w przypadku aplikacji nie bazujących na języku HTML. Standardy takie jak serwisy sieciowe (ang. *web-services*) czy XML bardzo często idą w parze ze stosunkowo prostym schematem programowania w serwerach WWW.

Wraz ze wzrostem popularności serwerów WWW pojawiła się potrzeba zapewnienia ich niezawodności, co wiąże się z koncepcją klastrów. W kolejnym punkcie zostaną naświetlone główne problemy, które mogą pojawić się przy implementowaniu klastra na potrzeby serwerów WWW.

2.3.1 Występujące problemy

Ze względu na specyfikę serwerów WWW należy przeanalizować cele jakie musi realizować klaster. W przypadku serwera (lub aplikacji) bezstanowej jedynym zadaniem klastra będzie równoważenie obciążenia i detekcja wadliwych węzłów, w celu zapobiegnięcia przesłaniu zadania do niesprawnej maszyny.

Jeżeli serwer, bądź zainstalowana na nim aplikacja, przechowują informację o swoim stanie (sesja użytkownika), to klaster musi zapewniać mechanizm odzyskiwania danych w przypadku awarii.

Spotyka się bardzo różne rozwiązania tych problemów w serwerach komercyjnych. Aby uzyskać niezawodność, w niektórych rozwiązaniach stosuje się zapis danych na nośnikach fizycznych, a w innych wykorzystuje się mechanizm replikacji danych pomiędzy węzłami klastra.

2.3.2 Stosowane rozwiązania

W dzisiejszych czasach w zasadzie każdy komercyjny serwer WWW może pracować jako klaster, co zostało wymuszone przez rynek. Dla osoby zajmującej się

projektem informatycznym od strony biznesowej słowa takie jak skalowalność czy odporność na awarie stały się wyznacznikiem jakości rozwiązań informatycznych. Każda licząca się na rynku firma stara się przekonać swoich klientów, że implementacja ich klastra jest najlepsza i zapewnia największą niezawodność. W pracy zostaną omówione dwa przykładowe rozwiązania zastosowane w komercyjnych serwerach wspierających standard J2EE (*Java 2 Enterprise Edition)*. Uwaga zostanie skupiona na mechanizmach zabezpieczeń zapobiegających utracie danych sesji użytkownika. Celowo zostanie pominięta część związana z EJB (*Entity Java Beans*) oraz JMS (*Java Messaging Service*), jako nie związana z pracą.

2.3.2.1 BEA Weblogic Server 8.0

Serwer Weblogic firmy Bea jest uznawany za najlepiej przystosowany do działania w klastrze serwer J2EE [1]. Autorzy klastra zaproponowali architekturę złożoną z maszyny równoważącej obciążenie (programowo lub sprzętowo) oraz połączonych siecią węzłów klastra z zainstalowanym serwerem Weblogic [10]. Architektura jest w pełni zdecentralizowana, to znaczy nie ma serwera, który nadzorowałby pracę całego klastra. Węzeł dołącza do klastra wysyłając w trybie rozgłoszeniowym komunikat w sieci powiadamiający pozostałe komputery o swoim istnieniu. Jeżeli któraś z maszyn przestanie wysyłać regularne komunikaty w trybie rozgłoszeniowym (tzw. *heartbeats*), to pozostałe uznają, że nastąpiła w niej awaria i przejmą jej zadania.

Każda sesja użytkownika posiada swój serwer macierzysty oraz serwer zastępczy. Identyfikatory obu węzłów klastra są zaszyte w nazwie ciasteczka (ang. *cookie*) określającego numer sesji użytkownika.

Domyślnie każde żądanie od klienta jest przekierowywane do macierzystego węzła sesji (czyli węzła, na którym sesja została utworzona). Wybór miejsca przekierowania następuje w serwerze rozdzielającym żądania. W przypadku awarii węzła macierzystego serwer równoważący przekierowuje żądanie do węzła zastępczego, który zawiera pełną kopię danych z sesji. Węzeł zastępczy przejmuje rolę macierzystego i wybiera inny węzeł jako zastępczy. Jednocześnie zmienia identyfikator sesji, aby kolejne żądania były kierowane do niego.

Niestety w przypadku awarii dwóch węzłów w sieci dane replikowane między nimi zostają bezpowrotnie utracone.

Drugą znaczącą wadą tego rozwiązania jest słabe równoważenie obciążenia w klastrze. Każdy użytkownik jest na stałe przypisany do konkretnego serwera i dopiero w momencie awarii zostaje przekierowany do innego. Czyli w ogólności nowo dodany węzeł w sieci nie przejmie obciążenia od pozostałych maszyn, a jedynie zacznie obsługiwać nowych użytkowników. W razie awarii węzła wszystkie jego sesje zostaną przejęte przez pozostałe maszyny, a po ponownym uruchomieniu nie będzie on w stanie z powrotem przejąć obciążenia. W praktyce może to doprowadzić do efektu domina: kolejny serwer przestaje odpowiadać z powodu coraz większej liczby obsługiwanych użytkowników zwiększanej wraz z każdą kolejną awarią.

Przy takim podejściu administrator systemu ma dużo mniej czasu na włączenie kolejnej maszyny do klastra, jeżeli zobaczy, że obciążenie zaczyna przerastać możliwości klastra. Jeżeli się spóźni, to użytkownicy, którzy zalogowali się przed dołączeniem nowego węzła nie będą w stanie komfortowo pracować.

2.3.2.2 Sybase Enterprise Application Server

Projektanci klastra z firmy Sybase [9] oferują dwa rozwiązania problemu przechowywania stanu sesji. Jedno z nich korzysta z bazy danych, a drugie, podobnie jak w serwerze Bea Weblogic, replikuje stan sesji pomiędzy dwoma wybranymi węzłami.

W pierwszym rozwiązaniu autorzy stworzyli system scentralizowany, w którym w środku klastra znajduje się węzeł z bazą danych zapisujący stan wszystkich sesji. W przypadku awarii jakiegoś węzła stan obsługiwanych przez niego sesji został wcześniej zapisany na fizycznym nośniku i może być odzyskany przez inne maszyny.

Niestety baza danych jest wąskim gardłem tego rozwiązania. Zapisanie informacji na fizycznym nośniku jest bardzo kosztowne (trwa dłużej niż przesłanie danych przez sieć). Zapis trzeba wykonywać po obsłudze każdego żądania co może doprowadzić do zatkania serwera kopii zapasowej i w rezultacie braku mechanizmu odzyskiwania danych po awarii.

Dodatkowo dochodzi tu problem wydajności silnika bazy danych – jeżeli zostanie stworzony klaster składający się z wielu maszyn i każda z nich będzie zapisywała stan swoich sesji w bazie, to może się okazać, że wydajność całości zależy nie od liczby węzłów w sieci, ale od możliwości serwera z bazą danych.

Niewątpliwą zaletą rozwiązania jest możliwość odzyskania stanu sesji nawet po awarii całego klastra (wszystkich węzłów, włącznie z bazą danych). Informacje są zapisane na fizycznym nośniku, więc zawsze można je odzyskać.

Drugie rozwiązanie jest bardzo podobne do rozwiązania firmy Bea.

3 Jakarta-Tomcat 5.0

Serwer Jakarta-Tomcat [8] jest darmowym serwerem WWW z ogólnodostępnym kodem źródłowym. Umożliwia on tworzenie dynamicznych stron w oparciu o standardy Java (serwlety i strony JSP). Pomimo bardzo prężnego rozwoju serwera wciąż brakuje w nim w pełni funkcjonalnego i działającego klastra. Twórcy Tomcata dostrzegli już jak duży wpływ na decyzję o wyborze kontenera aplikacji ma jego skalowalność i odporność na awarie. W wersji 5.0 wprowadzili standard klastra w kodach źródłowych serwera oraz podłączyli bardzo prosty moduł replikujący stan sesji między węzłami klastra. Przykładowa implementacja modułu nie rozwiązuje jednak wielu problemów, które mogą spowodować wadliwe działanie klastra. Stąd zrodziła się idea stworzenia nowego klastra dla serwera Jakarta-Tomcat.

W kolejnych punktach zostanie opisany serwer Jakarta-Tomcat oraz przedstawiona implementacja pierwszego modułu klastra napisana przez Filipa Hanika (por. p. 3.4).

3.1 Rozwój serwera Jakarta-Tomcat

Pierwszym pomysłodawcą i twórcą Tomcata był James Duncan Davidson (projektant i programista z firmy Sun). Pod koniec lat dziewięćdziesiątych rozpoczął on prace nad wzorcową implementacją kontenera dla serwletów i stron JSP. Po napisaniu sporej części systemu przekazał kody źródłowe organizacji Apache Software Foundation, aby dalszy rozwój serwera odbywał się pod jej patronatem. Organizacja ochrzciła projekt nazwą Jakarta-Tomcat i w 1999 roku opublikowała wersję 3.0, jako wtyczkę do serwera WWW – Apache. Kolejna wersja (4.0) była już w pełni niezależnym serwerem. Pojawiła się obsługa takich standardów jak JDBC, SSL czy Realms.

Serwer Jakarta-Tomcat stał się bardzo popularnym kontenerem serwletów oraz stron JSP, głównie za sprawą ogólnodostępnego kodu źródłowego oraz jego dobrej wydajności. Prosta implementacja ograniczonego podzbioru standardów J2EE [3] umożliwiła stworzenie bardzo szybkiego i „lekkiego” serwera, który mógł być bardzo dobrą alternatywą dla serwerów ze stronami pisanymi w języku PHP. W wielu projektach nie ma potrzeby wykorzystywania skomplikowanych mechanizmów EJB czy JMS, a do pełnej realizacji zadań wystarczą serwlety z możliwością dostępu do bazy danych. Do tego typu projektów idealnie nadawał się serwer Jakarta-Tomcat. Wiele projektów i firm korzysta z Tomcata jako jednego z komponentów, jak choćby Jonas [7] czy Hyperion Analyzer [5].

3.2 Główne założenia koncepcyjne

Serwer Jakarta-Tomcat jest przede wszystkim kontenerem serwletów (wersja 2.4) i stron JSP (wersja 2.0). Intencją twórców projektu nie było stworzenie pełnej implementacji standardu J2EE [3], ale szybkiego i stabilnego kontenera serwującego strony dynamiczne napisane w języku Java. Ponieważ cały serwer został napisany w języku Java, więc może działać w zasadzie na dowolnej platformie, posiadającej implementację maszyny wirtualnej.

W kolejnych wersjach doszło sporo różnego rodzaju dodatków, ułatwiających pracę programistom i administratorom systemu.

Autoryzacja

Doszło wsparcie dla obiektów autoryzacji (czyli tzw. *realms*). Administrator może stworzyć własne implementacje mechanizmu uwierzytelniania lub skorzystać z trzech gotowych komponentów (w szczególności dostępny jest mechanizm korzystający z bazy danych). Ponadto obsługę żądań można tunelować w protokole HTTPS.

Drzewa JNDI, JDBC, JavaMail

Tomcat udostępnia proste mechanizmy rejestrowania obiektów w drzewie nazw (JNDI). Programista może pobierać w kodzie obiekty, wyszukując je po nazwie. Najczęściej stosuje się ten mechanizm przy korzystaniu z połączeń do bazy danych (obiekty typu `javax.sql.DataSource`) – administrator może modyfikować parametry połączenia bez potrzeby zaglądania czy modyfikowania kodu aplikacji.

Analogicznie można korzystać z obiektów umożliwiających wysyłanie listów pocztą elektroniczną [4].

Menedżer bezpieczeństwa

Tomcat w wersji 5.0 posiada wsparcie dla menedżera bezpieczeństwa (ang. *security manager*). Administrator może definiować poziom izolacji przy wykonywaniu kodu napisanego przez programistów. W ten sposób można obronić się przed wykonaniem niebezpiecznego z punktu widzenia serwera kodu w aplikacjach, np.

```
System.exit(0);
```

co powodowałoby koniec pracy całego systemu.

3.3 Opis implementacji

Jakarta-Tomcat jest w całości napisany w języku Java.

Źródła systemu są podzielone na trzy grupy:

1. jakarta-tomcat-catalina – część, w której znajduje się faktyczna implementacja kontenera serwletów,
2. jakarta-tomcat-connectors – implementacja podstawowych typów połączeń stosowanych w systemie (między klientem a serwerem),
3. jakarta-tomcat-jasper – implementacja kompilatora do stron JSP.

Z punktu widzenia klastrów najciekawsza jest pierwsza część, ponieważ tu znajduje się implementacja jądra systemu, a także moduł do tworzenia klastra.

Tomcat został zaimplementowany w postaci hierarchicznego drzewa obiektów, w którym każdy węzeł może zostać przedefiniowany przez odpowiednie wpisy w plikach konfiguracyjnych. Takie podejście ułatwia modyfikację systemu, co miało duży wpływ na spopularyzowanie serwera.

3.3.1 Hierarchia obiektów

Serwer Jakarta-Tomcat zazwyczaj składa się z następującej hierarchii komponentów:

- Korzeniem drzewa jest maszyna (obiekt implementujący interfejs `org.apache.catalina.Engine`). Reprezentuje ona niezależny serwer.

- Maszyna posiada węzły (obiekty implementujące interfejs `org.apache.catalina.Host`), które reprezentują wirtualne węzły.
- Węzeł posiada konteksty (obiekty implementujące interfejs `org.apache.catalina.Context`), które reprezentują aplikacje internetowe (ang. *web applications*). Aplikacją może być plik z rozszerzeniem .war lub podkatalog w katalogu webapps.
- Kontekst składa się z serwletów zadeklarowanych przez programistę w pliku opisującym aplikację (plik web.xml) oraz menedżera sesji.
- Menedżer sesji (obiekt implementujący interfejs `org.apache.catalina.Manager`) zarządza sesjami użytkowników.

Taka hierarchia nie jest obligatoryjna – dla przykładu w sytuacji, gdy instalujemy Tomcat jako wtyczkę w serwerze Apache, drzewo degraduje się do ostatniego poziomu, czyli samych obiektów aplikacji. Wszystkie pozostałe stają się zbędne, gdyż obsługą żądania na wyższym poziomie zajmuje się macierzysty serwer.

Konfiguracja hierarchii obiektów znajduje się w pliku server.xml. Każdy poziom jest definiowany przez odpowiedni znacznik w języku XML. W szczególności, standardowa dystrybucja Tomcata dostarcza trzy różne rodzaje menedżerów sesji, które w zależności od zapotrzebowania można ustawiać w pliku konfiguracyjnym. Ze względu na rolę jaką odgrywa menedżer sesji w klastrze (przechowuje stan sesji), w p. 3.3.1.1 zostaną zaprezentowane dostępne implementacje tego obiektu.

3.3.1.1 Implementacje menedżera sesji

W standardowej dystrybucji serwera Tomcat dostępne są trzy rodzaje menedżera sesji:

1. standardowy (ang. *StandardManager*),
2. plikowy (ang. *PersistentManager*),
3. bazodanowy (ang. *JDBCManager*).

Pierwszy z nich udostępnia podstawową funkcjonalność menedżera sesji – czyli po prostu przechowuje dane w pamięci operacyjnej jednej maszyny.

Drugi pozwala na periodyczne zapisywanie stanu sesji do pliku i odzyskanie tych danych po restarcie maszyny. Jest również przydatny w sytuacji, gdy nie wszystkie sesje mieszczą się w pamięci operacyjnej maszyny. Wtedy menedżer zapewnia nam dodatkowy mechanizm wymiany.

Trzecie rozwiązanie działa analogicznie do drugiego z tą różnicą, że zapis odbywa się w bazie danych, w odpowiednio zdefiniowanym schemacie.

3.3.1.2 Klaster w serwerze Tomcat

W wersji 5.0 została dodana obsługa klastra. Administrator może zdefiniować klasę implementującą klaster (interfejs `org.apache.catalina.Cluster`) w znaczniku `Cluster`, w pliku konfiguracyjnym serwera. Klaster jest definiowany na poziomie węzła, dzięki czemu można łatwo rozdzielić aplikacje, które mają działać w trybie rozproszonym od tych działających na pojedynczym serwerze. Klaster jest odpowiedzialny za obsługę wszystkich aplikacji (kontekstów) zainstalowanych w obrębie węzła. Implementacja musi zapewnić mechanizmy replikacji i synchronizacji w obrębie grupy połączonych maszyn oraz może udostępniać metody instalowania i odinstalowywania aplikacji w całym klastrze (metody `installContext(String`

`contextPath, URL war)`, `start(String contextPath)` oraz `stop(String contextPath)`).

Dodatkowo rozszerzono specyfikację pliku opisującego aplikację (web.xml [3]) o znacznik `<distributable/>`, określający czy aplikacja ma być obsługiwana przez klaster. Jeżeli znacznik zostanie wstawiony, to system podłączy do kontekstu menedżer sesji utworzony przez implementację klastra. W przeciwnym przypadku do kontekstu zostanie podłączony standardowy menedżer.

Każdy obiekt tworzony w drzewie hierarchii Tomcata będzie miał skopiowane wartości parametrów z pliku konfiguracyjnego, poprzez wywołanie metod `setXXX(String value)` (gdzie XXX jest nazwą parametru oraz jednocześnie nazwą atrybutu w pliku XML). Dodatkowo w specjalny sposób traktowane są obiekty implementujące interfejs klasy `org.apache.catalina.Lifecycle`.

3.3.1.3 Obiekty klasy Lifecycle

Jeżeli tworzone na poziomie serwera obiekty implementują interfejs `org.apache.catalina.Lifecycle`, to w momencie startu systemu wywoływana jest dla nich metoda `start()`. Obiekty tego interfejsu zostaną powiadomione o zamknięciu systemu poprzez wywołanie dla nich metody `stop()`. Dla przykładu obiekt implementujący interfejs `Cluster` w module klastrowania jednocześnie implementuje interfejs `Lifecycle`, aby w metodach `start` i `stop` wykonać czynności przygotowujące do pracy oraz czynności kończące pracę klastra.

Każde żądanie obsługiwane przez serwer Tomcat przechodzi przez odpowiedni strumień wywołań procedur. Jest to najbardziej newralgiczne miejsce systemu, ze względu na częstość wywoływania jego kodu.

3.3.2 Strumień przetwarzania żądań

Strumień przetwarzania żądań w serwerze Tomcat składa się z następujących wywołań:

1. Strumień jest inicjowany przez konektor, dostarczający żądanie klienta.
2. Lokalizowany jest odpowiedni węzeł, a w nim kontekst, do którego odwołuje się żądanie.
3. Wykonywany jest ciąg wyzwalaczy (tzw. *valve*), które obudowują wywołanie serwletu (szerzej o wyzwalaczach będzie mowa w p. 3.3.3).
4. Uruchamiany jest serwlet.
5. Wyzwalacze kończą działanie.
6. Konektor przekazuje odpowiedź do klienta.

3.3.3 Wyzwalacze

Serwer Tomcat umożliwia podpięcie wyzwalaczy obudowujących wykonanie żądania przez serwlet. Zasada działania jest podobna do serwletów filtrujących [3], z tym że wyzwalacze definiuje się na poziomie węzła. Wyzwalacz musi być obiektem klasy implementującej interfejs `org.apache.catalina.Valve`.

W metodzie `invoke(Request, Response, Context)` programista może zdefiniować akcje, które będą wykonywane podczas obsługi każdego żądania. Programista decyduje czy żądanie ma być przetwarzane dalej, czy ma zostać przerwane, wywołując lub nie metody `invokeNext(Request, Response)`.

Przykładowo implementacja kontenera autoryzującego opiera się na wyzwalaczach. Przed wywołaniem odpowiedniej strony obsługi żądania sprawdzane jest czy klient posiada wystarczające uprawnienia.

Wyzwalacze umożliwiają tworzenie dzienników serwera lub stosowanie globalnych dla całego serwera filtrów przychodzących żądań.

W szczególności mechanizm ten jest również wykorzystywany przy implementacji klastra (patrz p. 3.4).

3.4 Implementacja klastra w serwerze Tomcat 5.0

W wersji 5.0 klaster dla serwera Tomcat został ustandaryzowany. Dołączono odpowiedni mechanizm podłączania i konfiguracji odrębnych implementacji mechanizmu rozpraszania obliczeń. Osobą odpowiedzialną za rozwój standardu, jak również dołączenie pierwszego w pełni działającego modułu klastra w oficjalnej dystrybucji serwera, jest Filip Hanik – członek zespołu programistów Tomcata.

Wybór Filipa Hanika jako osoby odpowiedzialnej za klaster nie był przypadkowy. Już wcześniej zaimplementował on działającą wtyczkę do Tomcata wersji 4.0 umożliwiającą stworzenie klastra.

3.4.1 Klaster dla wersji 4.0

Hanik zaimplementował w pełni funkcjonalny moduł dla Tomcata w wersji 4.0, który po podpięciu do serwera umożliwiał stworzenie klastra. Jego pomysł polegał na podmianie klasy menedżera sesji, w której dodał mechanizm replikowania zmian zachodzących w sesjach użytkowników. Jako warstwy transportującej użył biblioteki JavaGroups [6] dostarczającej mechanizmów ułatwiających programowanie w środowisku rozproszonym. Konfiguracja JavaGroups dopuszcza komunikację w trybie rozgłoszeniowym (ang. *multicast*), minimalizującą liczbę przesyłanych pakietów (niestety implementacja nie posiada mechanizmów zapewniających niezawodność transmisji, jaka jest w protokole TPC/IP).

Rozwiązanie Filipa Hanika było tylko zewnętrzną nadbudówką do serwera, który sam w sobie nie posiadał jeszcze wtedy żadnego wbudowanego wsparcia dla klastrów. Wymuszało to tworzenie odrębnych klastrów dla każdej aplikacji, co oczywiście powodowało zbędny narzut.

W wersji 5.0 moduł Filipa Hanika został przepisany i dołączony do oficjalnej dystrybucji.

3.4.2 Architektura klastra w wersji 5.0

Klaster jest całkowicie zdecentralizowany, każdy kolejny węzeł dołącza się rozsyłając odpowiedni komunikat w trybie rozgłoszeniowym, w lokalnej sieci. Sesje replikowane są do wszystkich węzłów – w szczególności zakłada się, że węzły posiadają identyczny stan każdej sesji. Klaster nie zakłada istnienia zintegrowanego mechanizmu równoważącego obciążenie – każde kolejne żądanie może trafić do dowolnego serwera w klastrze i będzie obsłużone tak, jakby cała interakcja odbywała się na jednej fizycznej maszynie. Takie podejście pozwala na zastosowanie dynamicznego równoważenia obciążenia, z czym wiąże się minimalizacja wariancji czasu obsługi żądania.

Klaster jest zaimplementowany jako moduł (plik z rozszerzeniem .jar), który można opcjonalnie dołączyć do serwera Tomcat.

3.4.3 Implementacja klastra w wersji 5.0

Kody źródłowe modułu można podzielić na trzy części:

1. warstwa transportująca (por. p. 3.4.3.1),
2. warstwa związana z serwerem Tomcat: menedżer sesji, obiekt sesji (por. p. 3.4.3.2),
3. klasy pomocnicze (por. p. 3.4.3.3).

3.4.3.1 Warstwa transportująca

Hanik zrezygnował z korzystania z biblioteki JavaGroups – jak sam twierdzi z powodu braku gwarancji poprawności przesyłanych informacji [11]. Zaimplementował warstwę opartą na protokole TCP/IP, służącą do wymiany danych między węzłami klastra. Każdy węzeł trzyma otwarte połączenia do wszystkich pozostałych węzłów. Lista pozostałych węzłów uaktualniana jest na podstawie periodycznie wysyłanych komunikatów o istnieniu węzła (w trybie rozgłoszeniowym). Jeżeli któryś węzeł przestanie wysyłać komunikat, to pozostałe komputery uznają, że nastąpiła w nim awaria i wyrzucą go z listy członków klastra. Niestety pojawia się tu problem rozspójnienia klastra – może zdarzyć się, że część węzłów uzna, że dany komputer przestał działać (np. z powodu nadmiernego obciążenia sieci, bądź procesora), a część pozostawi go na liście aktywnych węzłów. Wtedy podczas replikacji uaktualniona wersja sesji nie dotrze do wszystkich komputerów. Przy założeniu, że cały klaster posiada spójną wersję sesji, może okazać się to dużym zagrożeniem utraty danych. Wystarczy, że kolejne żądanie zostanie przekierowane do węzła nie posiadającego najnowszej wersji sesji.

Każdy węzeł przy starcie otwiera jeden port, na którym będzie oczekiwał na połączenia od pozostałych węzłów. Połączenie nawiązywane jest tylko raz, a przy wysyłaniu danych korzysta się z wcześniej otwartego połączenia. Niestety między każdymi dwoma węzłami otwierane są dwa osobne połączenia, co negatywnie wpływa na wydajność sieci.

Każdy węzeł otwiera jeden port w trybie rozgłoszeniowym, w celu periodycznego wysyłania komunikatu o swoim istnieniu. W komunikacie znajduje się informacja o węźle oraz o porcie, na którym nasłuchuje. Jeżeli odbiorca komunikatu nie posiada otwartego połączenia do ogłaszającego się węzła, to inicjowane jest nowe połączenie.

Wadą wysyłania komunikatów w trybie rozgłoszeniowym jest uniemożliwienie pracy dwóch serwerów Tomcat na jednej fizycznej maszynie, tak aby oba należały do tego samego klastra. Tylko jeden z serwerów będzie w stanie otworzyć konkretny port rozgłoszeniowy.

Wszelkie informacje wymieniane między węzłami klastra przesyłane są za pomocą wiadomości. Komunikacja jest całkowicie bezstanowa – to znaczy po wysłaniu komunikatu nadawca nie oczekuje na odpowiedź, minimalizując ryzyko potencjalnych zakleszczeń (ang. *deadlock*). Format rozsyłanych wiadomości jest następujący:

- 7 bajtów – preambuła,
- 1 bajt – długość wiadomości,
- dane wiadomości,
- 7 bajtów – zakończenie wiadomości.

Na dane składa się zserializowana postać klasy `SessionMessage`. Klasa zawiera typ wiadomości, identyfikator sesji, dane sesji, identyfikator kontekstu oraz adres węzła wysyłającego wiadomość.

Występują następujące typy wiadomości:

1. `EVT_SESSION_CREATED` – została utworzona nowa sesja lub istniejąca sesja została zmieniona;
2. `EVT_SESSION_EXPIRED_WONOTIFY` – wygasła sesja, ale nie należy powiadamiać o tym słuchaczy (ang. *listner*);
3. `EVT_SESSION_EXPIRED_WNOTIFY` – wygasła sesja i należy powiadomić słuchaczy;
4. `EVT_SESSION_ACCESSED` – użytkownik odwołał się do sesji, ale jej nie zmieniał;
5. `EVT_GET_ALL_SESSIONS` – nowy węzeł pobiera wszystkie do tej pory stworzone sesje;
6. `EVT_ALL_SESSION_DATA` – przesyłane są dane sesji.

W aktualnej wersji rozwiązania zdarzenia `EVT_SESSION_EXPIRED_WONOTIFY` oraz `EVT_SESSION_EXPIRED_WNOTIFY` są obsługiwane jednakowo, z powiadamianiem słuchaczy.

Zapisywanie i odtwarzanie danych sesji

Dane sesji są serializowane za pomocą wywołania standardowej metody `writeObjectData(ObjectOutputStream stream)` z klasy `StandardSession`, a deserializowane za pomocą metody `readObjectData(ObjectInputStream stream)`. Przy odtwarzaniu danych sesji z tablicy bajtów, tworzony jest strumień wejściowy `ReplicationStream`, który czyta obiekty korzystając z mechanizmu ładowania klas dostarczanego przez kontekst aplikacji, do której należy sesja. W ten sposób przesyła się obiekty klas stworzonych w ramach danej aplikacji.

3.4.3.2 Warstwa związana z serwerem Tomcat

Moduł Hanika podłączany jest za pomocą klasy `SimpleTcpCluster`, którą definiuje się w znaczniku `<Cluster>`, w pliku konfiguracyjnym serwera. Klasa implementuje standardowy interfejs `org.apache.catalina.Cluster`, dostarczając niezbędnych metod do pracy systemu. W aktualnej wersji implementacja nie wspiera metod instalacji oraz deinstalacji aplikacji w całym klastrze.

Klasa `SimpleTcpCluster` implementuje interfejs `org.apache.catalina.Lifecycle`, wykorzystując metody `start()` oraz `stop()` do inicjowania swoich struktur danych.

Przy starcie systemu w metodzie `start()` obiekt tworzy warstwę transportującą, przekazując do niej parametry pobrane z pliku konfiguracyjnego server.xml (dokładniej o parametrach klastra będzie mowa w p. 3.4.4).

W metodzie `createManager(String name)`, odpowiadającej za tworzenie menedżera sesji, przekazywany jest obiekt instancji klasy `SimpleTcpReplicationManager`, będącej nadklasą klasy `StandardManager`. Klasa przeimplementowuje metodę `createSession()`, w której tworzy i przekazuje obiekt

typu `ReplicatedSession` zamiast `StandardSession`. Obiekty replikowanych sesji przechwytują wywołania metod

- `setAttribute(String name, Object value)`,
- `removeAttribute(String name)`,
- `expire(boolean notify)`

z poziomu aplikacji w celu ustawienia bitu informującego o przeprowadzonych zmianach w sesji. Po zakończeniu przetwarzania żądania system podejmuje decyzję o replikacji na podstawie wartości tego bitu.

Inicjowanie procesu replikacji zachodzi w odpowiednim wyzwalaczu (obiekt klasy `ReplicationValve`), podpiętym pod wywołania żądań. Po wykonaniu żądania przez aplikację obsługującą dany kontekst, wyzwalacz wywołuje metodę `requestCompleted(String sessionId)` w obiekcie zarządcy sesji. Metoda przekazuje wiadomość, zawierającą zserializowane dane sesji, która zostaje rozesłana do wszystkich węzłów w klastrze.

Rozsyłanie wiadomości do węzłów może być wykonywane w dwóch trybach: synchronicznym oraz asynchronicznym. Przy pierwszym trybie wątek obsługujący żądanie jest równocześnie odpowiedzialny za rozesłanie pakietów w obrębie klastra. Powoduje to oczywiście wydłużenie czasu obsługi żądania, co może negatywnie wpływać na wydajność systemu.

W trybie asynchronicznym tworzone jest zadanie replikacji sesji, które zostanie wykonane przez jeden ze specjalnych wątków – zazwyczaj już po zakończeniu obsługi żądania. Przy takim podejściu klient nie musi niepotrzebnie czekać na zakończenie zadania replikacji sesji; proces ten wykonywany jest w tle, w czasie przesyłania odpowiedzi do klienta. Niestety implementacja tego mechanizmu ma dużą wadę – nie jest w żaden sposób sprawdzane czy węzeł zdążył rozesłać nową wersję sesji przy kolejnym odwołaniu. Czyli może zdarzyć się następująca sytuacja:

1. Klient wysyła żądanie do klastra i zostaje przekierowany do maszyny A.
2. Podczas obsługi żądania sesja zostaje zmodyfikowana (na maszynie A).
3. Klient otrzymuje odpowiedź i natychmiastowo wysyła kolejne żądanie.
4. Serwer równoważący obciążenie przekierowuje żądanie do maszyny B.
5. Kod obsługujący żądanie odwołuje się do sesji, której maszyna A nie zdążyła jeszcze wysłać do maszyny B.

Taka sytuacja może z łatwością zajść przy nadmiernym obciążeniu danego węzła. Z powodu braku mocy obliczeniowej (lub przeciążenia sieci) wysyłanie wiadomości ze zmienionym stanem sesji zaczyna się opóźniać, co może doprowadzić do rozspójnienia klastra i w konsekwencji utraty danych. W praktycznych zastosowaniach tryb asynchroniczny jest rozwiązaniem niedopuszczalnym, właśnie z powodu ryzyka utraty danych.

Kolejnym poważnym problemem implementacji jest brak synchronizacji dostępu do sesji w obrębie klastra. Klaster nie posiada żadnego mechanizmu kontrolującego równoczesny dostęp do tej samej sesji. Jeżeli klient wyśle równolegle dwa żądania, modyfikujące te same zasoby, to doprowadzi to do utraty danych. Co gorsze, może okazać się, że część węzłów będzie posiadała sesję zmienioną przez jedno żądanie, a część przez drugie – jeżeli wiadomości z replikami będą docierały w różnej kolejności.

Taka sytuacja może mieć miejsce w przypadku stron HTML posiadających ramki. Po odświeżeniu strony przeglądarka wysyła równolegle wiele żądań do tych samych zasobów, automatycznie generując równoległe odwołania do tej samej sesji.

3.4.3.3 Klasy pomocnicze

Buforowanie

Został zaimplementowany prosty mechanizm buforowania przesyłanych w sieci informacji. W przypadku pracy w trybie asynchronicznym po przetworzeniu żądania kolejkowane jest zadanie replikacji sesji. Jeżeli zadanie dotyczące tej samej sesji znajdowało się już w kolejce, to zostanie ono nadpisane przez nowszą wersję. W ten sposób eliminuje się niepotrzebny ruch w sieci. Niemniej jednak taka sytuacja ma miejsce tylko przy dużym obciążeniu sieci, kiedy węzeł nie jest w stanie wystarczająco szybko wysłać wiadomości do pozostałych węzłów. Niestety zysk z wykorzystania tego mechanizmu jest iluzoryczny, ponieważ jest mało prawdopodobne, że kolejne żądanie trafi do tego samego węzła. Skoro zakłada się, że mogą występować tak duże opóźnienia w rozsyłaniu replik zmienionej sesji, to klaster bardzo szybko stanie się niespójny i zacznie tracić dane.

Pula wątków

W celu minimalizowania narzutu związanego z tworzeniem wątków została zaimplementowana pula wątków, obsługująca przychodzące wiadomości. Jeżeli wątek nasłuchujący na otwartych połączeniach z pozostałymi węzłami odbierze dane, to budzi jeden z wątków z puli i przydziela mu gniazdo (ang. *socket*), z którego należy wczytać dane. Wątek wczytuje kolejne wiadomości przetwarzając je synchronicznie. Po zakończeniu zwraca gniazdo i przechodzi w stan oczekiwania.

3.4.4 Konfiguracja klastra w wersji 5.0

Klaster włącza się poprzez odkomentowanie sekcji klastra w pliku server.xml:

```
<Cluster
   className="org.apache.catalina.cluster.tcp.SimpleTcpCluster"
   name="FilipsCluster"
   debug="10"
   serviceclass="org.apache.catalina.cluster.mcast.McastService"
   mcastAddr="228.0.0.4"
   mcastPort="45564"
   mcastFrequency="500"
   mcastDropTime="3000"
   tcpThreadCount="2"
   tcpListenAddress="auto"
   tcpListenPort="4001"
   tcpSelectorTimeout="100"
   printToScreen="false"
   expireSessionsOnShutdown="false"
   useDirtyFlag="true"
   replicationMode="synchronous"
/>
```

oraz sekcji wyzwalacza wykorzystywanego przez klaster:

```
<Valve
  className="org.apache.catalina.cluster.tcp.ReplicationValve"
  filter=".*\.gif;.*\.js;.*\.jpg;.*\.htm;.*\.html;.*\.txt;"
/>
```

Znaczenie odpowiednich atrybutów w sekcji klastra:

1. `name` – nazwa klastra (wartość w zasadzie nie wykorzystywana),
2. `debug` – poziom szczegółowości generowanych przez implementację zapisów systemowych,
3. `serviceclass` – klasa służąca do dostarczania informacji o dostępnych węzłach w klastrze (musi implementować interfejs `org.apache.catalina.cluster.MembershipService`),
4. `mcastAddr` – adres rozgłoszeniowy, na którym węzły będą powiadamiały się nawzajem o swoim istnieniu,
5. `mcastPort` – port używany przy rozgłaszaniu,
6. `mcastFrequency` – częstotliwość rozsyłania informacji o istnieniu węzła (w milisekundach),
7. `mcastDropTime` – czas po jakim węzeł zostaje uznany za niesprawny od momentu otrzymania ostatniego pakietu o jego istnieniu,
8. `tcpThreadCount` – liczba wątków obsługujących przychodzące wiadomości w klastrze,
9. `tcpListenAddress` – adres, na którym węzeł spodziewa się połączeń od pozostałych węzłów (jeżeli ustawiona jest wartość „auto”, to zostanie użyty domyślny adres komputera). Wykorzystuje się go, jeżeli komputer posiada więcej niż jedną kartę sieciową,
10. `tcpListenPort` – port, na którym nasłuchuje węzeł,
11. `tcpSelectorTimeout` – czas w milisekundach po jakim wątek nasłuchujący na otwartych połączeniach ma sprawdzić czy serwer nie jest zamykany,
12. `printToScreen` – czy strumień diagnostyczny ma być przekierowany do standardowego wyjścia,
13. `expireSessionsOnShutdown` – czy podczas zamykania serwera sesje mają zostać zdezaktualizowane,
14. `useDirtyFlag` – czy sesja ma być replikowana tylko po wywołaniu metod `setAttribute(..)`, `removeAttribute(..)`, `expire()`, `invalidate()`, `setPrincipal(..)`, `setMaxInactiveInterval(..)`,
15. `replicationMode` – przyjmuje wartość `synchronous` lub `asynchronous` i oznacza tryb w jakim sesje będą replikowane.

Wyzwalacz przyjmuje atrybuty:

1. `filter` – zawiera wyrażenia regularne oddzielone znakiem średnika, definiujące adresy, podczas przetwarzania których na pewno sesja nie będzie zmieniana. Przy obsłudze tych żądań mechanizm replikacji nie będzie wywoływany, nawet gdy flaga `useDirty` będzie ustawiona na fałsz.

3.4.5 Zalety rozwiązania

Zaletą przedstawionego rozwiązania jest jego całkowite zdecentralizowanie. Klaster nie posiada wyróżnionego węzła, co zwiększa stabilność i odporność na

awarie konkretnej maszyny. Dołączanie kolejnego węzła wiąże się jedynie z włączeniem go do sieci.

Kolejną cechą wyróżniającą klaster w serwerze Tomcat jest całkowita niezależność od algorytmu równoważenia obciążenia. Rozwiązanie nie wymaga specjalnego oprogramowania interpretującego identyfikatory sesji czy zapamiętującego przypisania węzeł-sesja. Administrator może wykorzystać dowolny mechanizm przekierowywania (programowy czy sprzętowy).

Pełna replikacja (tzn. każdy węzeł replikuje swoje sesje do wszystkich pozostałych węzłów) zapewnia dużą odporność klastra na awarie. Wystarczy, że chociaż jeden z serwerów pozostanie sprawny, aby nie utracić żadnych informacji. Poza tym umożliwia zastosowanie wyrafinowanych algorytmów równoważenia obciążenia, które nie będą ograniczane stałymi przypisaniami sesji do węzłów. W szczególności po dodaniu kolejnego węzła do klastra bardzo szybko możemy przerzucić na niego obciążenie z pozostałych komputerów, a nie tylko przekierowywać nowo przybyłych użytkowników.

Niestety rozwiązanie oprócz zalet ma również wady. Niektóre z nich są na tyle poważne, że uniemożliwiają wykorzystanie serwera w komercyjnych zastosowaniach. Rozwiązanie nie gwarantuje poprawnego działania systemu.

3.4.6 Wady rozwiązania

Główną wadą rozwiązania jest brak synchronizacji przy dostępie do sesji oraz brak kontroli nad spójnością klastra. Założenie o pełnej replikacji pociąga za sobą konieczność zapewnienia mechanizmu synchronizacji przy wprowadzaniu zmian czy chociażby odczycie sesji. Jeżeli każdy komputer może uchodzić za serwer macierzysty dowolnej sesji (czyli może obsługiwać wszelkie żądania, jakie docierają do klastra), to należy uniemożliwić wprowadzanie równoczesnych zmian na różnych maszynach. Niestety rozwiązanie Filipa Hanika ignoruje zagrożenie równoległych zmian, tak samo jak ignoruje możliwość opóźnień w replikacji sesji. Jeżeli komputer z powodu nadmiernego obciążenia opóźni rozesłanie nowej wersji sesji, to przy kolejnym żądaniu klienta, przekierowanym do innego węzła, aplikacja będzie korzystała z nieaktualnej wersji danych, nadpisując tym samym poprzednie zmiany. Co gorsze w takiej sytuacji nie ma pewności, która wersja sesji trafi do pozostałych węzłów, ponieważ będzie to zależało tylko od kolejności w jakiej odbiorą one repliki z dwóch różnych źródeł.

Nie mniej ważne jest niebezpieczeństwo rozspójnienia klastra. Zakłada się, że wszystkie węzły w klastrze posiadają identyczną wersję dowolnej sesji, ale co się stanie jeżeli węzeł chwilowo uzna inny komputer za niesprawny i nie prześle mu repliki zmienionej sesji? Taka sytuacja może zaistnieć przy dużym obciążeniu sieci lub maszyny – wystarczy, że na czas nie dotrze do któregoś węzła pakiet rozgłoszeniowy informujący o istnieniu innego węzła. Ten odrzuci go, sądząc, że węzeł przestał działać i nie będzie wysyłał do niego wiadomości. Za chwile pakiet o istnieniu znowu dotrze i węzeł ponownie zostanie włączony do listy aktywnych członków, ale sesja nie zostanie już zaktualizowana.

Korzystanie z trybu rozgłoszeniowego w sieci uniemożliwia zainstalowanie dwóch węzłów klastra na jednej fizycznej maszynie – tylko jeden serwer otworzy port rozgłoszeniowy i będzie mógł z niego korzystać. To może okazać się sporą niedogodnością w niektórych topologiach klastrów. Czasami administrator może chcieć zainstalować więcej niż jeden serwer na fizycznej maszynie zapewniając

większą odporność klastra na awarie oprogramowania. Niestety przedstawione rozwiązanie wyklucza taki model.

Architektura rozwiązania powoduje generowanie dużego ruchu w sieci – każdy węzeł rozsyła repliki sesji do wszystkich pozostałych. Przy dużym obciążeniu sieć może okazać się wąskim gardłem całego klastra. Powoduje to spore ograniczenia na liczbę węzłów jednocześnie działających w klastrze. Liczba przesyłanych komunikatów jest kwadratowo zależna od liczby podłączonych węzłów.

Reasumując można stwierdzić, że Hanik przygotował bardzo stabilne podstawy do implementacji modułu klastra dla serwera Jakarta-Tomcat. Wyłonił się standard jaki muszą spełniać moduły, aby mogły poprawnie działać przy każdej kolejnej dystrybucji Tomcata oraz został naszkicowany schemat podłączenia modułu do serwera. Wzorcowa implementacja dostarcza wiele wskazówek, które mogą ułatwić pracę twórcom kolejnych rozwiązań. Niestety nie spełnia ona wymagań stawianych przez komercyjne zastosowania i może służyć jedynie jako przykład.

Celem tej pracy jest stworzenie modułu, bazującego na koncepcji Hanika, wykorzystującego zalety rozwiązania, ale przede wszystkim eliminującego jego wady i zagrożenia. Główną zaletą, która została zidentyfikowana w bazowym rozwiązaniu, jest możliwość dynamicznego sterowania obciążeniem poszczególnych jednostek klastra. Główną wadą jest brak synchronizacji podczas odwoływania się do sesji oraz brak kontroli nad spójnością klastra. Drugim bardzo ważnym celem jest optymalizacja rozwiązania, aby jego wydajność nie stała się powodem rezygnacji ze stosowania klastra w serwerze Jakarta-Tomcat. W rozdziale 4 przedstawię nową, autorską implementację klastra, a w rozdziale 5 opiszę wyniki testów wydajnościowych zaproponowanego rozwiązania.

4 Nowy klaster dla serwera Jakarta-Tomcat

W tym rozdziale przedstawiam w pełni funkcjonalny i sprawnie działający klaster dla serwera Jakarta-Tomcat. Nowy projekt bazuje na koncepcji replikacji „każdy do każdego", ale dodatkowo rozwiązuje większość problemów zidentyfikowanych w implementacji Hanika. W szczególności rozwiązuje problem synchronizacji przy dostępie do sesji oraz uniemożliwia przypadkowe rozspójnienie klastra. Nowa implementacja dostarcza pewniejszy mechanizm automatycznej detekcji i obsługi awarii.

W rozdziale zostanie przedstawiony projekt i założenia koncepcyjne nowego klastra. Zostanie również naświetlona implementacja ważniejszych części systemu, wraz z zastosowanymi algorytmami.

4.1 Architektura i założenia koncepcyjne

Architektura zbliżona jest do tej występującej w klastrze z Tomcat wersji 5.0 – każdy węzeł posiada połączenia do wszystkich pozostałych i zakłada się, że stan każdej sesji na każdym serwerze jest identyczny. Brak tutaj przypisania sesji do konkretnego serwera.

Aby zapewnić efektywne synchronizowanie dostępu do sesji konieczne stało się scentralizowanie klastra.

4.1.1 Koncepcja centralnego sterowania

Wybrany węzeł w sieci pełni rolę semafora przy dostępie do zasobów. Przechowuje on informacje o wszystkich dostępnych obiektach oraz węzłach, które oczekują na zwolnienie obiektów. Granulacja dostępu jest na poziomie węzeł-sesja, to znaczy cały obiekt sesji jest udostępniany w danej chwili tylko jednemu węzłowi. Jeżeli serwer obsługuje wiele żądań równocześnie odwołujących się do tej samej sesji, to aplikacja internetowa musi zapewnić synchronizację w obrębie pojedynczej maszyny wirtualnej serwera (np. poprzez wykorzystanie słowa kluczowego `synchronized` w języku Java). Jeżeli żądania zostaną rozrzucone po różnych maszynach, to zostaną wykonane synchronicznie w nieokreślonej kolejności, ale każde żądanie będzie wykonywane z aktualną wersją sesji. Przy takim rozwiązaniu aplikacje, w których wymagana jest równoczesna obsługa wielu żądań dotyczących tej samej sesji (np. z powodu synchronizowania wątków żądań), będą musiały wykonywać się w środowisku ze zintegrowanym mechanizmem równoważącym obciążenie (patrz p. 4.1.4). Mechanizm ten zapewni przekierowywanie zgłoszeń odwołujących się do tego samego zasobu na maszynę, która w danym momencie go zajmuje.

Stworzenie węzła wyróżnionego (serwer zarządca) umożliwiło implementację centralnego nadzorowania spójności klastra. Maszyna zajmująca się synchronizacją dostępu do sesji równocześnie jest odpowiedzialna za przechowywanie informacji o węzłach klastra i rozpropagowywanie tych informacji do każdego węzła. W ten sposób węzeł zostanie uznany za niesprawny dopiero wtedy, gdy serwer zarządca uzna go za niesprawny i także dopiero wtedy zostanie on odrzucony przez wszystkie pozostałe węzły. Takie podejście eliminuje niebezpieczeństwo zaistnienia sytuacji, gdzie jeden z węzłów z powodu utraty łączności odrzuca chwilowo inny węzeł, nie przesyłając mu uaktualnionej wersji sesji. Dopóki węzeł nie dostanie komunikatu od centralnego serwera o odrzuceniu danej maszyny, dopóty będzie próbował nawiązać

połączenie w celu ukończenia procesu replikacji, nie zwalniając replikowanych sesji. Taki mechanizm gwarantuje, że każde żądanie, które trafi do klastra będzie obsługiwane z najnowszą wersją sesji. W najgorszym przypadku mogą następować opóźnienia spowodowane nie zwalnianiem sesji przez węzły, ale to można wyeliminować, jeżeli rolę zarządcy przejmie serwer równoważący obciążenie. Wtedy każde kolejne żądanie będzie przekierowywane do serwera aktualnie zajmującego sesję, do której nastąpiło odwołanie.

Implementacja przewiduje stałą konfigurację, gdzie serwerem centralnym będzie z góry ustalony komputer. Niemniej jednak istnieje możliwość rozszerzenia klastra tak, aby serwer zarządca był wybierany drogą elekcji, wśród aktualnie podłączonych węzłów (patrz p. 6). Wtedy w momencie awarii głównej maszyny inny węzeł mógłby przejąć jego rolę i zachować ciągłość pracy systemu. W aktualnej wersji zaimplementowanego klastra, aby zapewnić ciągłość pracy w przypadku awarii serwera zarządcy niezbędne byłoby zastosowanie mechanizmu dublowania serwera (ang. *mirroring*). W sytuacji, gdy serwer synchronizacyjny przestałby odpowiadać nastąpiłoby automatyczne odłączenie komputera od sieci i wstawienie na jego miejsce identycznie skonfigurowanej maszyny (z tym samym adresem IP oraz serwerem synchronizacyjnym słuchającym na tym samym porcie). Wtedy pozostałe węzły ponownie nawiązałyby połączenia z serwerem zarządcą i rozpoczęłaby się procedura przyłączania kolejnych węzłów klastra (dokładniej jest to opisane w p. 4.2.4). W wyniku tych działań system zostałby samoistnie odbudowany.

4.1.2 Podział na warstwy

Implementacja modułu klastra składa się z dwóch podstawowych warstw:

1. warstwa transportująca,
2. warstwa synchronizacyjna.

Pierwsza z warstw jest odpowiedzialna za komunikację w obrębie klastra – tzn. nawiązuje połączenia z ustalonymi komputerami i umożliwia przesyłanie danych w sieci. Dodatkowo implementacja tej warstwy musi być odporna na tymczasowe awarie w sieci. Jeżeli zostanie zlecone wysłanie pakietu do danego komputera to, warstwa ma dotąd próbować wysłać dane, aż w całości dotrą one do odbiorcy lub do momentu, gdy komputer ten nie zostanie *explicite* wyrzucony z listy aktywnych adresów.

Warstwa synchronizacyjna jest ściśle związana z serwerem Tomcat, zapewniając mechanizmy wyłącznego dostępu do sesji. W sytuacji, gdy serwer pokrywa się z serwerem zarządcą, warstwa działa lokalnie. Jeżeli serwer synchronizujący jest na innej maszynie, to warstwa w sposób przezroczysty komunikuje się z zarządcą poprzez warstwę transportującą. System wykorzystuje warstwę w celu założenia blokady na zasoby związane z przetwarzanym żądaniem. Po zakończeniu przetwarzania blokada jest zwalniana. Warstwa przechowuje informacje o wszystkich dostępnych węzłach w klastrze i jest odpowiedzialna za replikację sesji.

W przypadku serwera zarządcy implementacja warstwy synchronizacyjnej jest dodatkowo rozbudowana. Przechowuje szczegółowe informacje o wszystkich dostępnych zasobach i kolejkuje żądania dostępu do tych zasobów. W momencie zwolnienia zasobu warstwa przydziela obiekt kolejnemu czekającemu węzłowi, wysyłając do niego odpowiedni komunikat.

4.1.3 Wersjonowanie sesji

W systemie każda sesja posiada swój numer wersji. Numer początkowo jest ustawiany na zero i jest zwiększany po przetworzeniu każdego kolejnego żądania odwołującego się do danej sesji. Numer wersji jest używany podczas odzyskiwania danych po awarii węzła. Jeżeli dany węzeł przetworzył żądanie zmieniające sesję i przed zwolnieniem blokady tej sesji nastąpiła awaria, to system wykorzysta numer wersji w celu wyszukania najnowszych danych wśród pozostałych węzłów. Istnieje szansa, że węzeł zdążył przesłać uaktualnioną replikę sesji do chociaż jednego z pozostałych serwerów i na tej podstawie uda się odzyskać wprowadzone zmiany. Po awarii węzła serwer zarządca dla każdej sesji, której węzeł nie zdążył zwolnić, wysyła zapytanie o numer wersji do pozostałych komputerów. Ten który przekaże najwyższy numer wersji rozpropaguje swoje dane do pozostałych. Oczywiście mechanizm nie zadziała w przypadku sesji, których obsługa nie została w pełni ukończona, jednak każda awaria pociąga za sobą pewne niebezpieczeństwo utraty danych, a zadaniem oprogramowania jest minimalizowanie tego ryzyka. Mechanizm odzyskiwania danych po awarii węzła został dokładniej opisany w p. 4.2.

4.1.4 Zintegrowany mechanizm równoważenia obciążenia

Każdy klaster musi posiadać serwer przekierowujący żądania do końcowych węzłów – czyli tzw. punkt dostępowy (serwer ruchu). Aplikacje klienckie zgłaszające się do systemu negocjują połączenie z punktem dostępowym, nie widząc architektury samego klastra. Serwer ten zazwyczaj pełni jednocześnie rolę jednostki równoważącej obciążenie, równomiernie dzieląc żądania na wszystkie maszyny. Posiada on swój stały adres IP, na który są tłumaczone adresy WWW w serwerach nazw[1]. Ponieważ serwer ruchu musi posiadać informacje o dostępnych węzłach w klastrze, aby móc przesyłać do nich żądania od klientów, naturalne wydaje się połączenie serwera ruchu z serwerem zarządcą. Przy takim rozwiązaniu administrator nie musi dodatkowo konfigurować serwera ruchu, ustawiając mu listę dostępnych węzłów. Dodatkowym atutem wykorzystania zintegrowanego mechanizmu równoważenia obciążenia jest możliwość zwiększenia wydajności systemu, poprzez skrócenie czasu reakcji na żądanie klienta. Serwer sterujący ruchem ma dostęp do informacji, który węzeł aktualnie zajmuje daną sesję. Posiadając takie dane jest w stanie kierować ruch tak, aby minimalizować liczbę stosunkowo drogich wywołań blokowania i odblokowywania sesji. Przy obsłudze zgłoszenia klient nie będzie musiał oczekiwać na zakończenie procesu replikacji sesji i zwolnienie blokady przez poprzednio obsługujący go węzeł, tylko od razu zostanie przekierowany do węzła aktualnie posiadającego najnowsze dane.

Zastosowanie takiego rozwiązania daje ogromne możliwości usprawnienia samego procesu replikacji. Mając pewność, że nikt nie będzie oczekiwał na zwolnienie blokady, każdy z serwerów może opóźniać moment oddania sesji, a co więcej może opóźniać moment replikacji. Dla zapewnienia niezawodności w przypadku awarii, każdy z serwerów replikuje zmienioną sesję, zaraz po zakończeniu obsługi żądania, do jednego, losowo wybranego węzła klastra. Zadania rozesłania danych do pozostałych węzłów zostają opóźnione. Jeżeli w międzyczasie, przed rozesłaniem zmian w sesji, nadejdzie kolejne żądanie zmieniające tę samą sesję, to

[1] Istnieje możliwość równoważenia obciążenia na poziomie serwerów nazw, poprzez przekazywanie różnych adresów IP dla tego samego adresu WWW, ale takie rozwiązanie nie zapewnia mechanizmu automatycznej naprawy awarii i jest dużo bardziej statyczne w kontekście równomiernej dystrybucji ruchu.

system w momencie wykonywania zadań replikacji roześle wersję uwzględniającą najnowsze zmiany. Zmniejsza to ruch w sieci, eliminując konieczność rozesłania wcześniejszych zmian. Bardzo ważne jest tutaj dobranie odpowiednich czasów buforowania, aby nie doprowadzić do sytuacji, gdzie żadna sesja nie zmienia kontekstu wykonania (analogicznie do architektury serwer macierzysty-zastępczy) i zanika główna zaleta architektury „każdy do każdego", czyli równomierne równoważenia obciążenia.

4.1.5 Bezpieczeństwo

System nie zapewnia żadnych mechanizmów bezpieczeństwa podczas replikowania sesji czy autoryzacji dołączanego węzła. Zakłada się, że klaster komunikuje się w prywatnej sieci, za ścianą ogniową (ang. *firewall*) i wszystkie odebrane komunikaty pochodzą od właściwych adresatów. Wszelkie przypadkowe lub złośliwe komunikaty w sieci mogą doprowadzić do wadliwego działania klastra, a nawet do jego całkowitego zablokowania. W gestii administratora systemu jest odpowiednie zabezpieczenie sieci, w której komunikują się węzły.

4.2 Działanie systemu

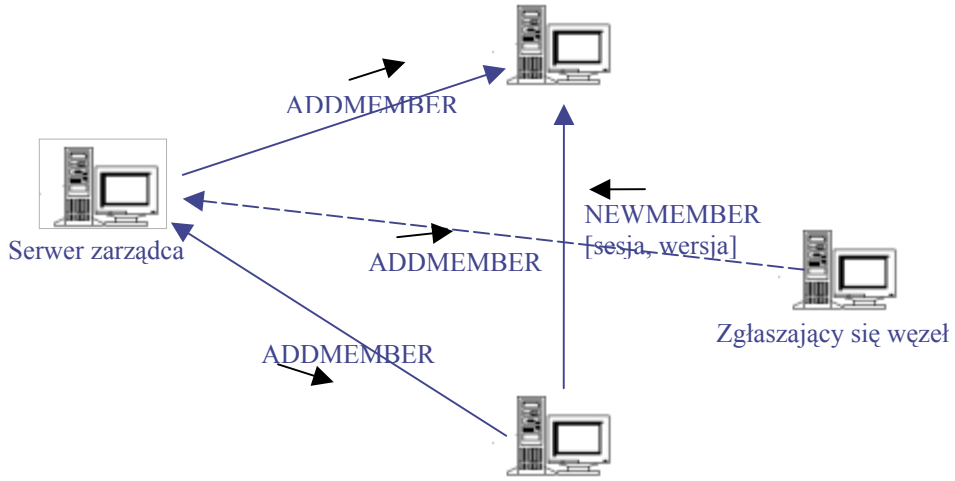
W tym podrozdziale zostanie opisany sposób reakcji systemu na podstawowe rodzaje zdarzeń, które mogą się pojawić podczas działania. Zostanie omówiony mechanizm dołączania nowego węzła czy reakcja na zerwanie połączenia i całkowitą awarię jednego z komputerów. Przy każdym zdarzeniu główny nacisk kładziony jest na zapewnienie niezawodności i minimalizowanie prawdopodobieństwa utraty danych przechowywanych w sesji.

Legenda diagramów:

	Pojedynczy węzeł klastra. Fizyczna maszyna może posiadać wiele odrębnych węzłów. Każdy węzeł to serwer Tomcat z nowym modułem wspierającym klastry.
	Serwer zarządca – może to być albo jeden z węzłów klastra, albo serwer ruchu.
	Węzeł, w którym nastąpiła awaria.
	Połączenie TCP/IP między dwoma węzłami. Ponieważ między każdymi dwoma węzłami jest nawiązywane tylko jedno fizyczne połączenie, więc kierunek strzałki oznacza stronę aktywną nawiązywanego połączenia.
	Nowo nawiązane połączenie TCP/IP. W przypadku diagramów ukazujących awarię węzła przerywana linia będzie oznaczała logiczne połączenie.
	Zerwane fizyczne połączenie (np. z powodu zerwania kabla lub awarii karty sieciowej).
KOMUNIKAT	Dużymi literami oznaczane są przesyłane komunikaty. Strzałka nad komunikatem informuje o kierunku przesyłania.

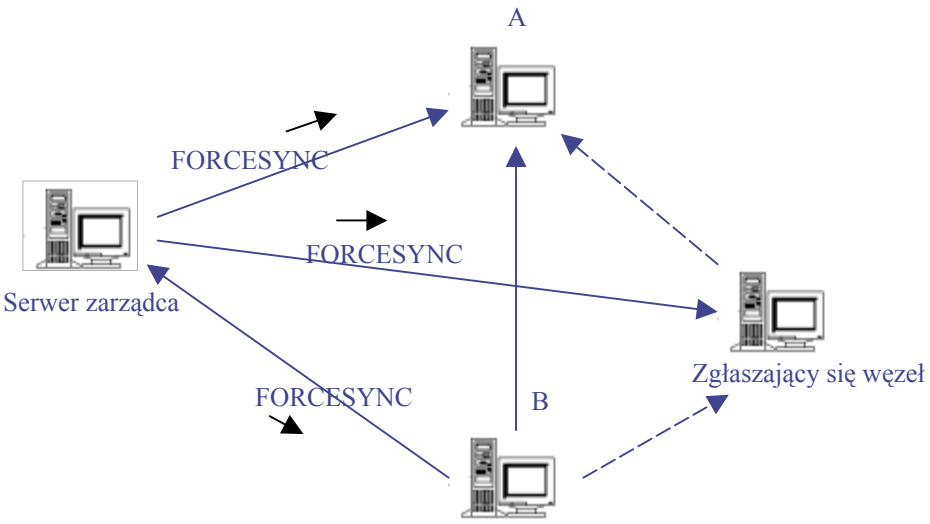
4.2.1 Dołączanie nowego węzła

Jest to standardowa sytuacja, która ma miejsce przy każdym restarcie któregokolwiek z węzłów klastra lub też podczas rozbudowy superkomputera.



Rys. 4.1. Schemat zgłaszania się nowego węzła w klastrze

Zgłaszający się węzeł nawiązuje połączenie z serwerem zarządcą (por. rys. 4.1); zakładamy, że węzeł zna adres IP oraz port serwera zarządcy (ustawione na sztywno w pliku konfiguracyjnym lub pobrane z sieci poprzez protokół rozgłoszeniowy). Wysyła komunikat `NEWMEMBER`, wraz z identyfikatorami wszystkich posiadanych sesji oraz numerami wersji. Serwer zarządca, po odebraniu komunikatu wysyła komendę `ADDMEMBER` do wszystkich pozostałych węzłów, informującą o konieczności przyłączenia nowego węzła (komunikat zawiera adres nowego węzła). Do zgłaszającego się węzła wysyła komunikat `ADDMEMBER` z adresami przyłączonych do tej pory węzłów. Czyli w pierwszej fazie serwer zarządca uspójnia klaster, wymuszając tworzenie nowych par połączeń. Bez komunikatu `ADDMEMBER` od serwera zarządcy żaden z węzłów nie może przyjąć nowych połączeń.



Rys. 4.2. Schemat wykonywanych operacji po przyłączeniu węzła

W drugiej fazie następuje synchronizacja zasobów (por. rys. 4.2). Dla każdego identyfikatora sesji, wysłanego przez nowy węzeł, serwer zarządca sprawdza numer wersji z numerem zapisanym u siebie w strukturach danych. Są możliwe trzy sytuacje:

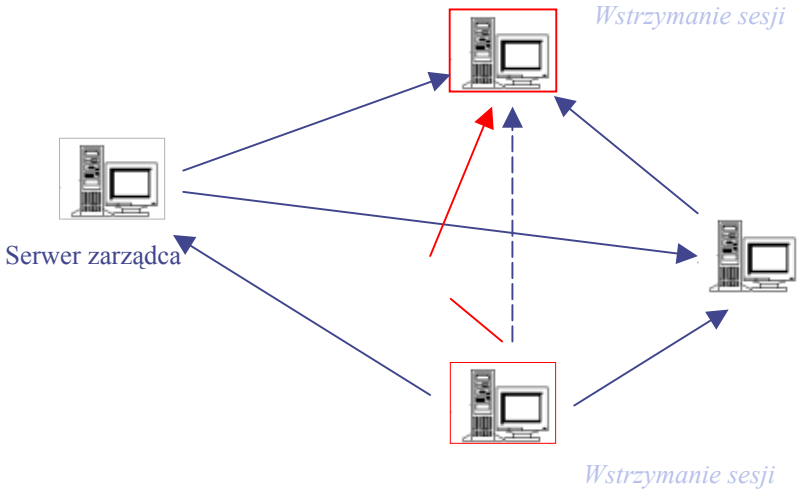
1. Numery wersji są identyczne – nie są podejmowane żadne działania.
2. Numer wersji trzymany przez zarządcę jest mniejszy od numeru dostarczonego przez nowy węzeł – serwer wysyła do nowego węzła komunikat `FORCESYNC` z identyfikatorem tej sesji.
3. Numer wersji trzymany przez zarządcę jest większy od numeru dostarczonego przez nowy węzeł – serwer wysyła komunikat `FORCESYNC` do jednego z wcześniejszych członków klastra. Komunikat zawiera identyfikator sesji oraz adres nowego węzła.

Komunikat `FORCESYNC` powoduje wymuszenie replikacji sesji dla danego identyfikatora. Opcjonalnie można wysłać adres węzła, do którego należy wysłać replikę. Jeżeli adres nie zostanie podany, to replika będzie rozesłana do wszystkich członków klastra. Dodatkowo serwer zarządca przed wysłaniem komunikatu ustawia adresata jako aktualnie zajmującego sesję. Jest to konieczne, ponieważ serwer zarządca zakłada, że wszystkie podłączone węzły posiadają identyczny stan wszystkich sesji. Jeżeli nie zostałaby założona blokada, to żądanie odwołujące się do niezsynchronizowanej sesji mogłoby zostać przekierowane do węzła „A”, który jeszcze nie otrzymał nowej repliki. Wtedy węzeł założyłby blokadę w serwerze synchronizacyjnym i zaczął przetwarzanie żądania. Niestety żądanie miałoby dostęp do nieaktualnej wersji sesji. Mechanizm obroni się przed taką sytuacją, uniemożliwiając węzłowi „A” założenie blokady dopóty, dopóki nie zwolni jej węzeł replikujący. Ten z kolei zwolni sesję dopiero po zakończeniu procesu replikacji, czyli przesłaniu najnowszej wersji danych do węzła „A”.

W sytuacji, gdy podłączany węzeł jest całkowicie nowym serwerem Tomcat, przesyłana tablica identyfikatorów sesji będzie pusta. Niemniej jednak przy ponownym podłączaniu tego samego węzła (na przykład z powodu chwilowego zerwania połączenia) tablica może zawierać dane.

4.2.2 Awaria połączenia

Awaria połączenia może nastąpić z winy karty sieciowej (przepalenie układu scalonego), z powodu przerwania kabla lub błędu w konfiguracji sieciowej systemu operacyjnego.



Rys. 4.3. Schemat działania klastra w przypadku awarii połączenia

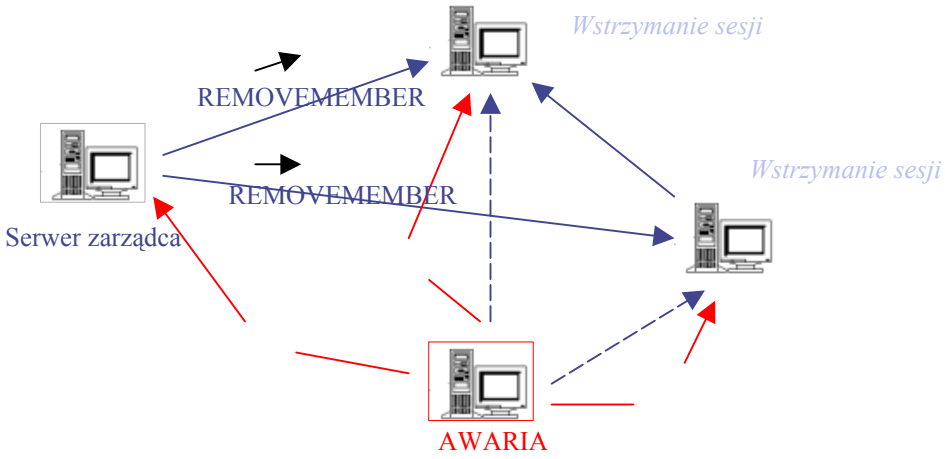
Jeżeli mamy do czynienia z awarią połączenia między dwoma zwykłymi węzłami klastra (tzn. nie dotyczy ona serwera zarządcy), to nie spowoduje ona żadnych zmian w architekturze klastra (por. rys. 4.3). Węzły będą trzymały logiczne połączenie w swoich strukturach danych, nie zaprzestając prób wysyłania komunikatów replikacyjnych między sobą. Jedyną różnicą będzie to, że komunikatu w rzeczywistości nie będzie udawało się wysłać i węzły będą wstrzymywały zajęte przez siebie sesje (sesja jest zwalniana tylko w momencie poprawnego rozesłania replik do wszystkich członków klastra). Przy wykorzystaniu zintegrowanego serwera ruchu końcowy użytkownik nawet się nie zorientuje, że nastąpiła jakakolwiek awaria połączeń. Jeżeli nie wykorzystamy zintegrowanego serwera ruchu, to żądanie odwołujące się do zajętej sesji może trafić do innego węzła i jego wykonanie zostanie opóźnione o czas naprawy zerwanego połączenia.

Przy tej okazji łatwo zauważyć konieczność trzymania blokady sesji do momentu zakończenia procesu replikacji. Gdyby węzeł wcześniej zwolnił blokadę, to kolejne żądanie odwołujące się do sesji mogłoby trafić do węzła, z którym zostało zerwane połączenie, a ten rozpocząłby przetwarzanie na nieaktualnych danych.

Jeżeli mamy do czynienia z awarią połączenia między serwerem zarządcą i zwykłym węzłem, to zostanie ona potraktowana jako awaria węzła. Dokładniejszy opis podejmowanych działań znajduje się w p. 4.2.3.

4.2.3 Awaria węzła

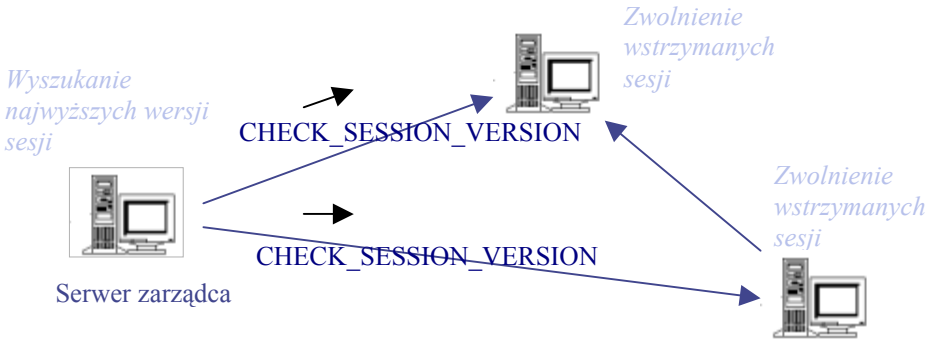
Opisywana sytuacja może zdarzyć się w momencie awarii fizycznej maszyny, na której pracuje serwer węzła lub w przypadku braku odpowiedzi ze strony programu serwera (por. rys. 4.4).



Rys. 4.4. Schemat działania klastra w przypadku awarii węzła

W przypadku całkowitej awarii jednego ze zwykłych węzłów klastra zakłada się, że zostaną zerwane wszystkie połączenia fizyczne (w szczególności połączenie z serwerem zarządcą). Na tej podstawie serwer zarządca stwierdza fakt awarii.

Po zerwaniu połączenia zarządca rozsyła komunikat `REMOVEMEMBER`, wraz z adresem uszkodzonej maszyny, do wszystkich pozostałych członków klastra. Każdy węzeł po otrzymaniu tego komunikatu ma prawo wyrzucić komputer z puli aktywnych członków i anulować wszystkie zadania wysłania komunikatów do tego komputera. Czyli *de facto* może zakończyć wstrzymane procesy replikacji i tym samym przestać zajmować do tej pory pobrane sesje.



Rys. 4.5. Schemat działania klastra po odrzucenia węzła

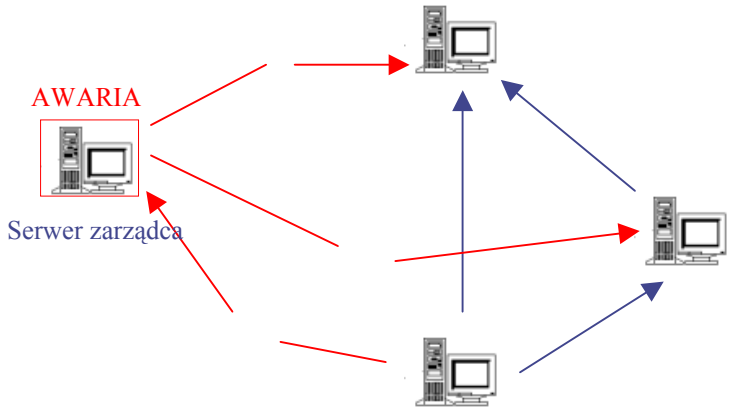
Dodatkowo serwer zarządca próbuje odzyskać najnowsze wersje wszystkich, zajmowanych przez uszkodzoną maszynę sesji (por. rys. 4.5). Dla każdego identyfikatora z zajętych sesji rozsyła komunikat `CHECK_SESSION_VERSION` do wszystkich członków z prośbą o przesłanie trzymanego przez węzeł numeru wersji.

Po zebraniu tych informacji, jeżeli okaże się, że któryś z węzłów posiada wyższy numer niż ten przechowywany u zarządcy, serwer synchronizacyjny wysyła komunikat `FORCESYNC` do posiadacza najwyższego numeru, tym samym odzyskując zmienione dane.

Wykorzystując ten mechanizm zwiększamy szanse odzyskania danych utraconych przy awarii węzła. Może zdarzyć się, że maszyna zdążyła wysłać komunikat replikacyjny do chociaż jednego z węzłów, umożliwiając odzyskanie zmienionego stanu sesji. Taka sytuacja może pojawić się przy wykorzystaniu zintegrowanego serwera ruchu oraz zwiększeniu czasu buforowania wysyłanych replik. Wtedy każdy węzeł zaraz po zakończeniu przetwarzania żądania wysyła replikę sesji do jednej, losowo wybranej maszyny, a do pozostałych wysyła z pewnym opóźnieniem. Jeżeli opóźnienie będzie odpowiednio długie może okazać się, że do wszystkich węzłów roześle replikę dopiero po przetworzeniu kilku żądań, tym samym ograniczając liczbę przesyłanych pakietów w sieci. Stąd bierze się potrzeba odzyskiwania stanu nie zwolnionych sesji. Szerzej ten temat jest opisany w p. 4.3.4.

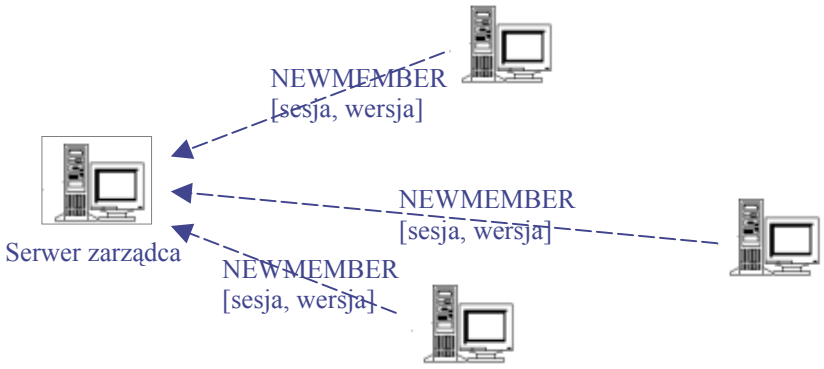
4.2.4 Awaria serwera zarządcy

Opisywana sytuacja może zdarzyć się w momencie awarii fizycznej maszyny, na której działa serwer zarządca lub w przypadku braku odpowiedzi ze strony programu serwera – rys. 4.6.



Rys. 4.6. Awaria serwera zarządcy

W przypadku awarii serwera zarządcy każdy z węzłów likwiduje wszelkie połączenia z pozostałymi węzłami, uznając że klaster przestał funkcjonować. Dalsze zachowanie każdego z węzłów jest identyczne z sytuacją przyłączania nowego członka do klastra. Każdy z serwerów próbuje połączyć się z zarządcą. Udaje im się to dopiero po restarcie maszyny zarządcy (lub ewentualnie wpięciu identycznie skonfigurowanej maszyny zapasowej).



Rys. 4.7. Schemat odzyskiwania struktury klastra po awarii serwera zarządcy

Wysłany zostaje komunikat `NEWMEMBER`, zawierający tablicę par: sesja wraz z numerem wersji (por. rys. 4.7). Dalsze kroki podejmowane przez system zostały opisane w p. 4.2.1.

Niestety serwer zarządca podczas przyjęcia komunikatu `NEWMEMBER` wraz z wersjami sesji posiadanych przez węzeł, nie jest w stanie rozstrzygnąć czy podane wersje są aktualnie najwyższymi w klastrze. Taką wiedzę może zweryfikować dopiero po podłączeniu wszystkich węzłów, które wcześniej tworzyły klaster. Z tego powodu, przy implementacji serwera zarządcy, konieczne staje się chwilowe zablokowanie przetwarzania jakichkolwiek żądań po starcie serwera, aby pozwolić mu na zebranie informacji, niezbędnych do prawidłowej pracy systemu.

Przy takim rozwiązaniu awaria serwera zarządcy nie powoduje utraty danych. Oczywiście może spowodować, że system będzie przez pewien czas niedostępny, ale zakłada się, że taka sytuacja będzie miała miejsce stosunkowo rzadko, więc nie powinna negatywnie wpływać na wygodę pracy użytkowników aplikacji.

4.3 Implementacja

Mimo bardzo podobnej architektury i koncepcji rozwiązania przy implementacji w zasadzie nie udało się wykorzystać kodu napisanego przez Hanika. W skład nowego modułu weszło jedynie kilka plików źródłowych z poprzedniej implementacji, zawierających mechanizm podłączenia modułu do serwera. W szczególności są to klasy:

- `org.apache.catalina.cluster.session.SimpleTcpReplicationManager`,
- `org.apache.catalina.cluster.session.ReplicatedSession`,
- `org.apache.catalina.cluster.tcp.ReplicationValve`.

Nakład pracy jaką należałoby włożyć w celu dostosowania kodów źródłowych starego modułu do celów nowego projektu byłby porównywalny ze stworzeniem całości od nowa. Dlatego bardziej sensowne okazało się zaimplementowanie całego modułu od początku.

Moduł został napisany w języku Java (wersja 1.4), przy użyciu standardowych bibliotek, dostarczanych wraz z instalacją środowiska wykonywalnego Javy lub z dystrybucją serwera Jakarta-Tomcat. Wybór języka programowania został zdeterminowany przez technologię serwera Tomcat. Ponieważ cały system został zaimplementowany w języku Java, nie było powodu, aby wprowadzać inny język.

Kod modułu był pisany pod kątem maksymalizowania wydajności. Ponieważ zazwyczaj w tego typu systemach wąskim gardłem staje się przepustowość sieci, główna część uwagi podczas tworzenia projektu była poświęcona kwestii minimalizowania ilości przesyłanych informacji. Wiąże się to z wyeliminowaniem zbędnych pakietów (por. p. 4.3.1, 4.3.2) oraz z mechanizmami buforowania (por. p. 4.3.4). Dodatkowym utrudnieniem podczas pisania było silnie zrównoleglone środowisko pracy modułu. Serwer WWW pracuje wielowątkowo, obsługując nawet kilkaset żądań jednocześnie. Dobrze napisany moduł klastra musi być zabezpieczony przed równoległym dostępem, jednocześnie pozwalając na wykonywanie możliwie jak największej liczby operacji w tym samym czasie. Zbyt mała liczba rozłącznych sekcji krytycznych spowolni działanie modułu, tworząc wąskie gardła. Z kolei zbyt częste wywołania wejścia i wyjścia z sekcji krytycznej może negatywnie wpływać na wydajność i utrudniać wykrywanie błędów.

Dodatkową ważną kwestią, decydującą o sposobie budowania modułu była potrzeba działania w dwóch różnych trybach:

1. Jako klient, zgłaszający się do serwera zarządcy.
2. Jako serwer zarządca.

Zakłada się, że w drugim trybie moduł może pracować zarówno wewnątrz serwera Tomcat, z wykorzystaniem części klienckiej mechanizmu synchronizacji, jak i w niezależnej aplikacji (por. p. 4.3.7), wykorzystującej tylko część serwerową mechanizmu. Dokładniejszy opis zastosowanego rozwiązania znajduje się w p. 4.3.6.

W dalszej części pracy opisuję sposób implementowania poszczególnych części systemu, zaczynając od mechanizmu transportowania danych w sieci, a kończąc na algorytmie synchronizacji oraz równoważenia obciążenia. W celu uproszczenia notacji został zastosowany skrót `org..` oznaczający pakiet `org.apache.catalina.cluster`.

4.3.1 Warstwa transportująca

Ponieważ nowy klaster ma być rozwiązaniem tanim i łatwym w instalacji, jako mechanizm transportujący został wykorzystany protokół TCP/IP, przy użyciu którego komputery mogą się komunikować niemalże w każdej dostępnej sieci. Drugim bardzo ważnym czynnikiem, który spowodował wybór tej technologii jest powszechność gotowych, sprawdzonych implementacji (obsługa TCP/IP jest zaimplementowana w standardowych klasach Javy). Charakterystyka architektury klastra (połączenia każdy z każdym) sugerowałaby wykorzystanie protokołu rozgłoszeniowego, niemniej jednak brak mechanizmów gwarantujących dostarczenie danych oraz sprawdzenia ich poprawności, spowodował wykluczenie tego rozwiązania z projektu. Do poprawności działania klastra niezbędna jest gwarancja dostarczenia danych w niezmienionej postaci. Taką gwarancję daje protokół TCP/IP.

Na implementację warstwy transportującej składają się następujące klasy:

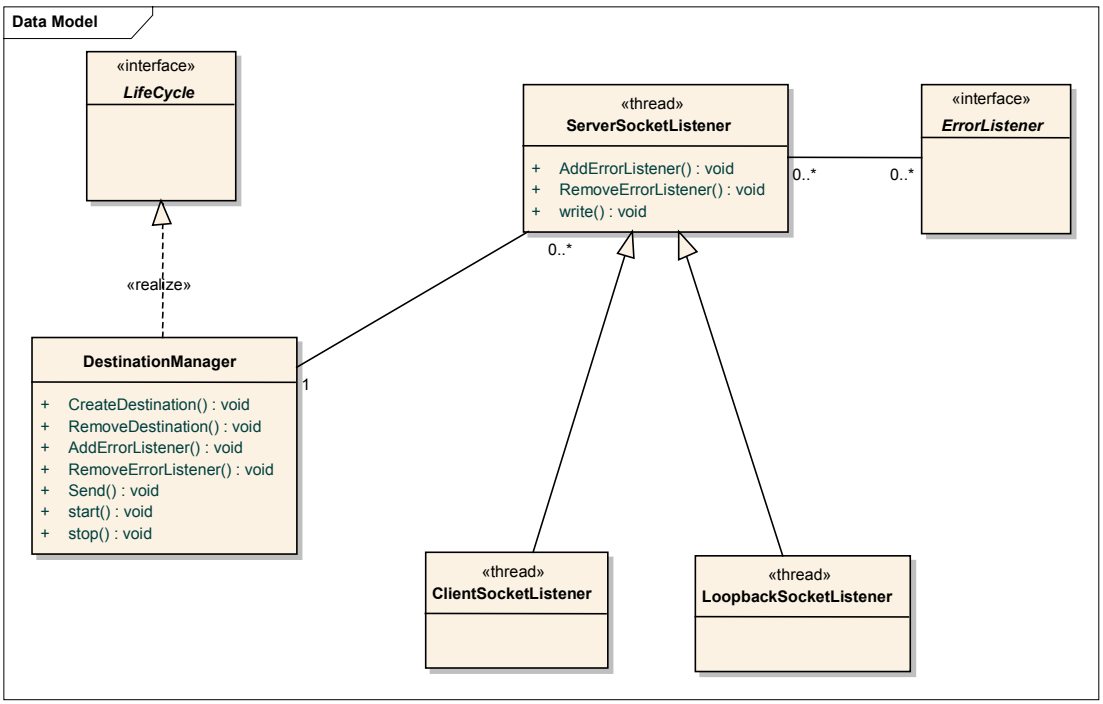
1. `org..transport.DestinationManager`,
2. `org..transport.DestinationAddress`,
3. `org..transport.socket.ServerSocketListener`,
4. `org..transport.socket.ClientSocketListener`,
5. `org..transport.socket.LoopbackSocketListener`.

Kluczową rolę w mechanizmie transportującym odgrywa klasa `DestinationManager`. Zawiera ona metody umożliwiające dodawanie/odejmowanie adresów dostępnych komputerów oraz umożliwia odbieranie i wysyłanie danych.

Każdy serwer w klastrze identyfikowany jest poprzez unikatowy klucz. Na klucz składa się adres IP oraz numer portu. Klasa `DestinationAddress` reprezentuje adresy serwerów w module, jednocześnie implementując interfejs `org..Member`. W klasie została nadpisana metoda `equals()`, która sprawdza zgodność adresu IP oraz portu porównywanych obiektów. Poza tym została zaimplementowana metoda `compareTo(DestinationAddress)`, w której porównywane są adresy IP lub w przypadku ich równości numery portów. Metoda ta jest wykorzystywana podczas wybierania strony aktywnej/biernej przy nawiązywaniu połączenia (zostanie to dokładniej opisane w dalszej części punktu).

Obowiązkiem obiektu `DestinationManager` jest obsługa wszelkich problemów związanych z połączeniem i przesyłaniem danych. Sporą wadą protokołu TCP/IP jest łatwość zerwania połączenia – każde połączenie jest uznawane za aktywne dopóki są otrzymywane pakiety. Niestety wystarczy, że z powodu przeciążenia sieci lub systemu operacyjnego pakiety zaczną się znacząco opóźniać i połączenie może zostać zerwane. Ponieważ w zastosowanej architekturze spójność klastra jest kontrolowana za pomocą aktywności połączeń, aby ograniczyć liczbę omyłkowych decyzji o awarii węzłów, implementacja warstwy transportującej umożliwia przezroczyste odtwarzanie chwilowo zerwanych połączeń. Po utracie połączenia system przez pewien czas próbuje odtworzyć kanał komunikacyjny; jeżeli się to nie powiedzie, to informuje wszystkich zarejestrowanych słuchaczy o utracie adresata. Mechanizm powiadamiania o utracie połączenia jest wykorzystywany przez klaster do podejmowania decyzji o awarii węzłów. W przypadku zwykłego węzła znaczenie ma tylko informacja o zerwaniu połączenia z serwerem zarządcą – węzeł musi zerwać wszystkie pozostałe połączenia i próbować odnowić komunikację z zarządcą. Z kolei w przypadku serwera zarządcy informacja o zerwaniu połączenia powoduje podjęcie decyzji o jego odrzuceniu z klastra. Ważną cechą obiektu klasy `DestinationManager` jest fakt, że dany adresat będzie uznawany przez obiekt jako aktywny dopóki nie zostanie *explicite* wywołane `RemoveDestination(DestinationAddress)`. Ten fakt ułatwia implementację mechanizmu replikacji – węzeł wysyła dane do wszystkich pozostałych węzłów, niezależnie od tego czy są one dla niego dostępne czy nie. Obiekt `DestinationManager` po prostu będzie bez końca ponawiał próbę wysłania lub stwierdzi fakt poprawnego zakończenia, jeżeli w międzyczasie adresat zostanie usunięty z listy aktywnych.

Aby uniknąć aktywnego oczekiwania na grupie połączeń, każdy kanał komunikacyjny ma swój własny wątek, który go obsługuje. Istnieją trzy typy połączeń: `ServerSocketListener`, `ClientSocketListener` oraz `LoopbackSocketListener`. Diagram na rys. 4.8 przedstawia hierarchię klas.



Rys. 4.8. Hierarchia klas warstwy transportującej

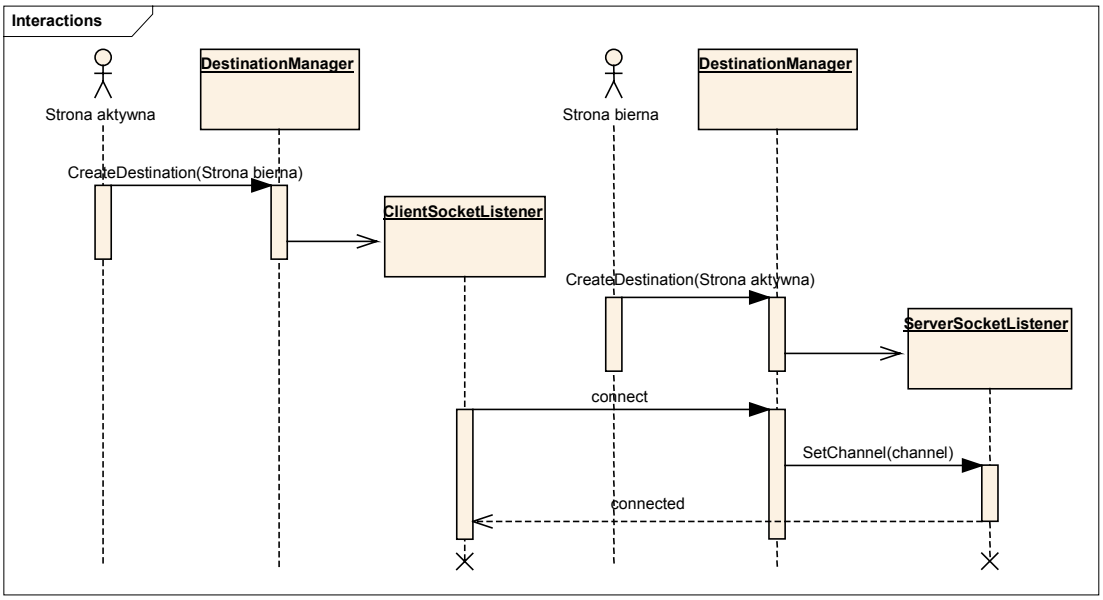
W celu zminimalizowania liczby przesyłanych pakietów w sieci, pomiędzy każdymi dwoma węzłami otwierane jest tylko jedno połączenie. Wyróżnia się stronę bierną i aktywną. Strona bierna, czyli obiekt klasy `ServerSocketListener`, nasłuchuje na porcie identyfikującym węzeł, oczekując na zainicjowanie połączenia przez stronę aktywną. Strona aktywna, czyli obiekt klasy `ClientSocketListener`, nawiązuje komunikację łącząc się na odpowiedni port adresata. Ponieważ strona bierna nie może zidentyfikować strony aktywnej (nie jest w stanie poznać numeru portu, na którym nasłuchuje strona aktywna, a jedynie adres IP, z którego się łączy), więc węzeł inicjujący połączenie wysyła w pierwszej kolejności numer swojego portu. Rola w połączeniu jest ustalana na podstawie wartości, którą przekaże metoda `compareTo(DestinationAddress)`. Poniższy kod jest wykonywany podczas tworzenia połączenia w obiekcie `DestinationManager`:

```
if (oAddr.compareTo(oLocalAddr) < 0)
   oCL = new ServerSocketListener(oAddr, this);
else if (oAddr.compareTo(oLocalAddr) > 0)
   oCL = new ClientSocketListener(this.oLocalAddr, oAddr, this);
else
   throw new RuntimeException("Will not add localhost");
```

Do prawidłowego działania systemu bardzo istotne jest, aby obie strony w tym samym momencie zaprzestały prób odtworzenia zerwanego połączenia. Sytuacja, w której strona aktywna dochodzi do wniosku, że połączenie jest zerwane, natomiast strona bierna wciąż oczekuje na odtworzenie kanału komunikacyjnego, może doprowadzić do błędnego działania systemu. Strona aktywna mogłaby przedsięwziąć pewne kroki związane z zerwaniem połączenia, a następnie ponowić próbę

nawiązania komunikacji. Wtedy strona bierna odtworzy połączenie, niestety bez powiadamiania słuchaczy o zerwaniu.

Schemat procesu nawiązywania połączenia jest przedstawiony na rys. 4.9.



Rys. 4.9. Diagram nawiązywania połączenia

Obiekt `DestinationManager` podczas tworzenia otwiera port do nasłuchu. W momencie próby nawiązania połączenia (powrót z metody `accept()`), menedżer sprawdza czy istnieje obiekt konektora obsługujący adres zgłaszającego się komputera. Jeżeli istnieje, to wywołuje na nim metodę `SetChannel(socket)`, tym samym kończąc procedurę nawiązywania połączenia. Jeżeli nie istnieje, to wywołuje na obiekcie klasy `MainSyncServer` (dokładniej o tej klasie będzie mowa w p. 4.3.6), metodę `CanAccept(DestinationAddress)`. Jeżeli metoda przekaże wartość pozytywną, to menedżer wywołuje sam dla siebie metodę `CreateDestination` z nowym adresatem jako argumentem. W tym momencie cała procedura nawiązywania połączenia zaczyna się na nowo. Taki mechanizm umożliwia serwerowi zarządcy akceptowanie połączeń z nowymi, nieznanymi wcześniej węzłami.

Jedyna różnica w konektorach biernym i aktywnym występuje w metodzie `Connect()`. Konektor bierny w tej metodzie przechodzi w stan oczekiwania na wywołanie metody `SetChannel(SocketChannel)`, natomiast konektor aktywny nawiązuje połączenie i wysyła jako pierwsze 4 bajty swój numer portu. Oba konektory w przypadku przechwycenia jakiegokolwiek wyjątku wywołują metodę `ReConnect()`, która próbuje odnowić zerwane połączenie.

Poza dwoma podstawowymi typami połączeń zostało jeszcze stworzone połączenie zwrotne, obiekt klasy `LoopbackSocketListener`. Wszystkie wysyłane przez niego dane od razu przekierowywane są do funkcji obsługi wiadomości sieciowych, nie wywołując niepotrzebnie funkcji systemowych. Jest on wykorzystywany podczas przesyłania danych od serwera zarządcy do serwera Tomcat w przypadku, gdy serwer Tomcat jest jednocześnie serwerem zarządcą. Architektura rozwiązania wymusiła taki mechanizm – serwer zarządca nie wie nic o istnieniu

serwera Tomcat. Jego zadanie sprowadza się jedynie do odbierania i wysyłania odpowiednich komunikatów. Takie podejście umożliwia odseparowanie serwera zarządcy od Tomcata (por. p. 4.3.7).

4.3.1.1 Odbieranie danych z sieci

Każdy konektor posiada swoją pulę buforów, do których może wpisywać pobrane z sieci informacje (maksymalna liczba dostępnych buforów jest konfigurowalna). Po odebraniu danych z sieci konektor pobiera kolejny wolny bufor, przepisuje do niego wiadomość i wykonuje na obiekcie `DestinationManager` metodę:

`AddReadyBuffer(DestinationAddress oFromAddr, ExtendableByteBuffer oData)`.

Po zakończeniu obsługi wiadomości na obiekcie `ExtendableByteBuffer` musi zostać wywołana metoda `ReturnToQueue()`, aby konektor mógł ponownie wykorzystać bufor do odebrania danych. Jeżeli liczba wolnych buforów spadnie do zera, to moduł wstrzyma odbiór danych, tym samym chroniąc system przed nadmiernym zużyciem pamięci.

Implementacja warstwy transportującej jest bardzo silnie zależna od protokołu TCP/IP. Przenośność i uogólnianie części transportującej nie były głównymi czynnikami decydującymi o kształcie końcowego kodu. Najważniejsza była wydajność tej części modułu, ponieważ to od niej zależała wydajność całego projektu. Takie zabiegi jak tworzenie pojedynczych połączeń, osobne wątki dla każdego kanału komunikacyjnego czy kolejki buforów miały na celu optymalizację modułu klastra. Mechanizm odtwarzania połączeń miał na celu zabezpieczenie klastra przed przypadkowym rozbiciem oraz ułatwienie implementacji samego mechanizmu replikowania sesji.

4.3.2 Komunikacja w klastrze

Komunikacja w klastrze odbywa się za pomocą warstwy transportującej. Aby możliwe było przesłanie danych konieczne jest wcześniejsze utworzenie adresata. W celu optymalizacji został stworzony model komunikacji bezstanowej – to znaczy wysłany komunikat zawiera pełen zbiór informacji potrzebnych do jego przetworzenia (analogicznie do modelu komunikacji w serwerach HTTP). Nie było potrzeby tworzenia skomplikowanych w implementacji dialogów między stronami. Przesyłane wiadomości mają następujący format:

- 32 bity – długość przesyłanej wiadomości.
- 8 bitów – typ wiadomości.
- Kolejne bity zawierają dane zależne od typu wiadomości.

Typy wiadomości można podzielić na dwa zbiory:

1. Wiadomości przesyłane między dwoma zwykłymi węzłami.
2. Wiadomości przesyłane między zwykłym węzłem i serwerem zarządcą.

Wiadomości przesyłane między dwoma zwykłymi węzłami

- `SESSIONS_ADD` – dodanie nowej lub modyfikacja istniejącej sesji. W sekcji danych znajduje się identyfikator sesji, kontekst oraz zserializowane dane sesji.

Wiadomości przesyłane między zwykłym węzłem i serwerem zarządcą

Wiadomości przesyłane od serwera zarządcy do zwykłego węzła:

- `DESTINATIONS_ADD` – dodanie nowego węzła do klastra. W sekcji danych przesyłany jest adres nowego węzła.
- `DESTINATION_REMOVE` – usunięcie węzła z klastra. W sekcji danych przesyłany jest adres węzła.
- `OBTAINED_SESSION` – sesja została przyznana węzłowi. W sekcji danych znajduje się identyfikator sesji, kontekst oraz aktualny numer wersji (jeżeli węzeł posiada mniejszy numer oznacza to, że z jakiś powodów nie otrzymał repliki ostatnich zmian i musi wstrzymać modyfikację sesji do czasu otrzymania najświeższych danych).
- `SESSION_EXISTANCE` – informacja czy podana sesja istnieje. W sekcji danych znajduje się identyfikator sesji, kontekst oraz 1 bajt z informacją czy sesja istnieje w klastrze czy nie. Komunikat jest wysyłany w odpowiedzi na `CHECK_SESSION_EXISTANCE`.
- `CHECK_SESSION_VERSION` – prośba o odesłanie numeru wersji sesji o podanym identyfikatorze. W sekcji danych przesyłany jest identyfikator sesji oraz kontekst. Komunikat jest wysyłany w momencie próby odzyskania sesji zajmowanych przez węzeł, który doznał awarii. Każdy węzeł w klastrze odsyła numer wersji, a serwer zarządca na tej podstawie wybiera węzeł, który zreplikuje swoją kopię danych do pozostałych członków.
- `SESSIONS_REMOVE` – usunięcie sesji. W sekcji danych znajduje się tablica identyfikatorów sesji oraz kontekstów. Komunikat jest wysyłany w momencie wygaśnięcia sesji. Serwer zarządca podejmuje decyzję o zakończeniu życia sesji analogicznie do pojedynczego serwera Tomcat. Jeżeli przez odpowiednio długi okres żaden serwer nie poprosi o dostęp do sesji, to zostaje ona uznana za nieaktywną.
- `SESSIONS_REPLICATE_WITH_SYNC` – wymuszenie procesu replikacji. W sekcji danych znajduje się adres węzła, do którego należy wysłać replikę (jeżeli jest pusty, to należy rozesłać repliki do wszystkich węzłów) oraz tablica identyfikatorów sesji, które należy replikować. Węzeł, który odbierze komunikat po zakończeniu replikacji musi zwolnić wszystkie sesje, ponieważ serwer zarządca przed wysłaniem wiadomości ustawia blokady na adresata. Mechanizm ten zabezpiecza przed niebezpieczeństwem wprowadzania zmian w sesjach przez inny węzeł w czasie replikowania. Wysłane repliki mogłyby nadpisać nowo wprowadzone zmiany.

Wiadomości przesyłane od zwykłego węzła do serwera zarządcy:

- `OBTAIN_SESSION` – zajęcie sesji. W sekcji danych znajduje się identyfikator zajmowanej sesji oraz kontekst.
- `RELEASE_SESSION` – zwolnienie sesji. W sekcji danych znajduje się identyfikator zwalnianej sesji, kontekst, numer wersji sesji oraz liczba zwolnień (zazwyczaj 1). Może się zdarzyć, że węzeł posiadając blokadę na sesji otrzyma od serwera zarządcy komunikat `SESSIONS_REPLICATE_WITH_SYNC` (na przykład z powodu zgłoszenia się nowego węzła do klastra). Wtedy węzeł musi zwolnić sesję więcej niż raz, wysyłając w tym celu liczbę zwolnień.

- `NEW_SESSION_MEMBER` – zgłoszenie się nowego węzła w klastrze. W sekcji danych znajduje się słownik par: identyfikator sesji (wraz z kontekstem), numer wersji. Komunikat jest wysyłany za każdym razem, gdy nastąpi zerwanie połączenia między zwykłym węzłem a serwerem zarządcą. W przypadku zupełnie nowego węzła słownik będzie pusty.
- `SESSION_CREATED` – utworzenie sesji. W sekcji danych znajduje się identyfikator sesji oraz kontekst. Komunikat jest wysyłany zaraz po utworzeniu sesji. Wysłanie komunikatu jest konieczne, aby pozostałe węzły mogły się dowiedzieć poprzez komunikat `CHECK_SESSION_EXISTANCE` o istnieniu tej sesji zanim twórca zakończy proces replikacji.
- `CHECK_SESSION_EXISTANCE` – sprawdzenie istnienia sesji. W sekcji danych znajduje się identyfikator sesji oraz kontekst. Komunikat jest wysyłany, jeżeli węzeł otrzyma żądanie odwołujące się do nieistniejącej sesji. Ponieważ sesja mogła być stworzona na innym węźle, ale jeszcze nie doszła replika, serwer wpierw sprawdza u zarządcy czy identyfikator jest aktywny.

Po odebraniu komunikatu z sieci tworzone jest zadanie (więcej o zadaniach będzie mowa w p. 4.3.3):

- `org..task.SessionReceivedTask` – w przypadku, gdy moduł działa wewnątrz serwera Tomcat (zadanie tworzone jest w obiekcie `org..transport.TransportCluster`).
- `org..task.MessageReceivedTask` – w przypadku, gdy moduł działa jako niezależny serwer zarządca (zadanie tworzone jest w obiekcie `org..transport.DestinationManager`).

Zadanie `SessionReceivedTask` dziedziczy po `MessageReceivedTask` dokładając kilka możliwych typów wiadomości, które może obsługiwać. W szczególności są to wiadomości odbierane przez zwykłe węzły od serwera zarządcy lub wiadomości wymieniane między zwykłymi węzłami. Zakłada się, że serwer zarządca nie musi nic wiedzieć o menedżerze sesji. Natomiast `SessionReceivedTask` posiada atrybut wskazujący na menedżera sesji – niezbędne podczas przetwarzania takich komunikatów jak chociażby `SESSIONS_ADD`.

Zadania tworzone na potrzeby przetwarzania komunikatów posiadają specjalną kolejkę zadań, gwarantującą, że komunikaty będą przetwarzane w kolejności, w której nadeszły (lub ewentualnie równolegle).

4.3.3 Mechanizm kolejki zadań zależnych

Na potrzeby projektu została stworzona wyspecjalizowana kolejka do przetwarzania zadań. Kolejka posiada swoją własną pulę wątków przetwarzających instrukcje. W ten sposób istnieje możliwość zlecania zadań, które mają się wykonać asynchronicznie (jak np. replikacja sesji). Wątki tworzone są raz, w momencie tworzenia kolejki, a ich praca sprowadza się do oczekiwania na nadejście kolejnego zadania, które mogłyby wykonać. Jeżeli zadań nie ma, to wątki przechodzą w stan uśpienia. Zmieniając liczbę wątków w kolejce można kontrolować zużycie zasobów systemowych przez moduł klastra (szerzej o konfigurowaniu klastra będzie mowa w p. 4.4).

Kolejka zadań posiada dodatkowe, bardzo przydatne możliwości:

1. definiowania zadań, które zostaną wykonane z opóźnieniem (np. zwolnienie sesji przy wykorzystaniu buforowania);
2. definiowania zależności między zadaniami – czyli zadanie zostanie wystartowane dopiero po wykonaniu innych zadań (np. zwolnienie sesji po zakończeniu wykonywania zadań replikacji);
3. restartowania zadania, czyli ponownego wrzucenia zadania do kolejki, w przypadku błędu (np. ponawianie prób replikowania sesji do danego węzła).

Każde zadanie musi dziedziczyć po abstrakcyjnej klasie `org..task.Task`. Klasa zawiera w szczególności abstrakcyjną metodę `InternalRun()`, w której podklasy umieszczają kod zadania. Klasa zawiera metodę `AddListener(Listener)`, poprzez którą można dołączyć słuchaczy do zadania. Słuchacze implementują interfejs `org..task.Listener`:

```
public interface Listener {
   public void WakeUp(boolean bError);
}
```

W momencie zakończenia wykonywania zadania dla każdego dołączonego słuchacza wywoływana jest metoda `WakeUp(boolean bError)`, z argumentem informującym czy zadanie zakończyło się z błędem czy nie (błąd jest wynikiem zgłoszenia przez zadanie wyjątku). Ten mechanizm jest wykorzystywany przy implementacji zależności między zadaniami. Po prostu klasa `Task` implementuje interfejs słuchacza, a w metodzie `WakeUp(boolean)` zmniejsza licznik zadań, na które oczekuje. W momencie, gdy licznik spadnie do zera, zadanie samo się inicjuje, dodając się do kolejki:

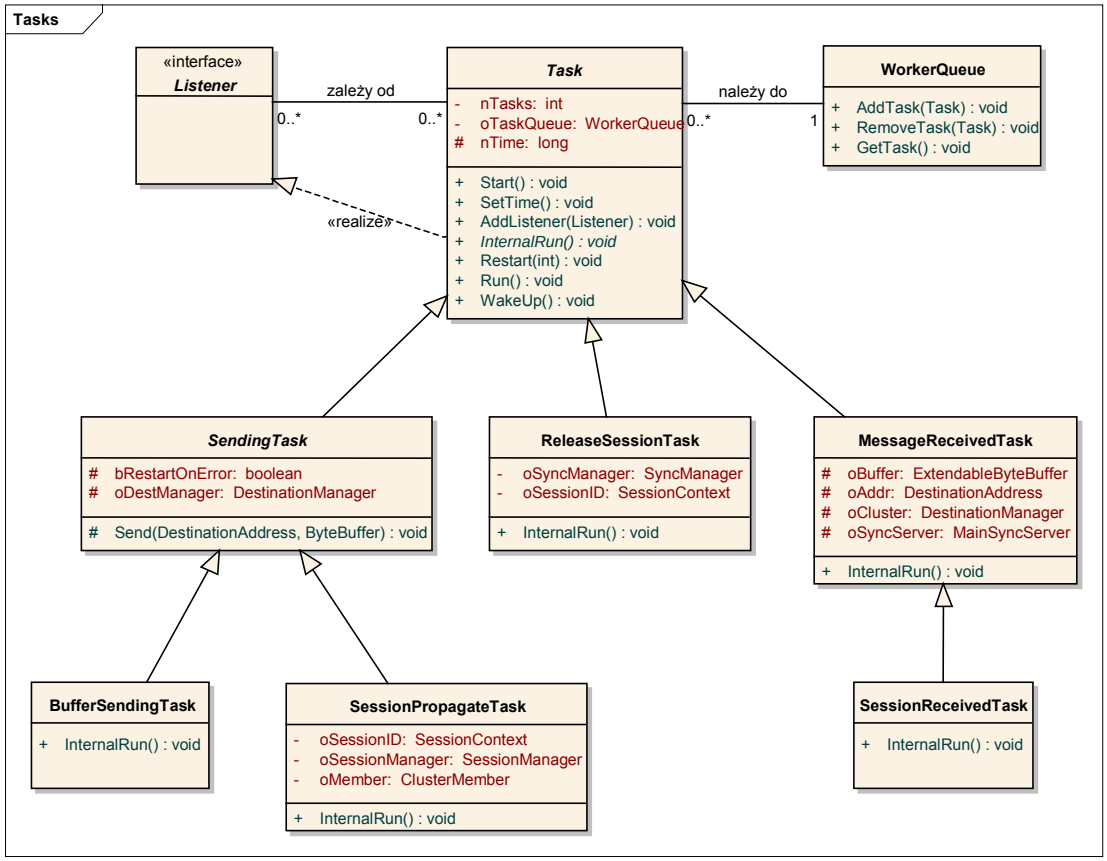
```
public void WakeUp(boolean bError) {
   this.DecTasks();
}

public synchronized void DecTasks() {
   nTasks--;
   Start();
}

public void Start() {
   if (nTasks <= 0) {
      if (this.nTimeoutFromStart > 0)
            this.SetTime(System.currentTimeMillis() +
                  nTimeoutFromStart);
      oTaskQueue.AddTask(this);
   }
}
```

Jeżeli zadanie ma się wykonać tylko w przypadku poprawnego zakończenia wszystkich zadań zależnych, to należy przeimplementować w podklasie metodę `WakeUp(boolean)`, zmniejszając licznik zależności tylko w przypadku braku błędu. Ten mechanizm został wykorzystany przy implementacji zadania zwalniania sesji (klasa `org..task.ReleaseSessionTask`):

```
public void WakeUp(boolean bError) {
   if (! bError)
      super.WakeUp(bError);
}
```

Zadanie zwolnienia sesji zależy od zadań replikacji sesji do poszczególnych węzłów. Może się ono wykonać tylko w przypadku poprawnego zakończenia wszystkich zadań replikacyjnych (co jest równoznaczne z poprawnym rozesłaniem replik do wszystkich węzłów w klastrze). Jeżeli wystąpi nieoczekiwany błąd podczas wykonywania zadań replikujących (np. brak pamięci w systemie), to sesja nie zostanie zwolniona, co zabezpieczy system przed groźbą utraty danych.

Na rys. 4.10 znajduje się hierarchia klas najważniejszych zadań zdefiniowanych w systemie.



Rys. 4.10. Diagram hierarchii zadań

Opis zadań:

1. `SendingTask` – abstrakcyjna klasa, zawierająca metodę `Send(DestinationAddress, ByteBuffer)`, która umożliwia wysłanie bufora z danymi do konkretnego adresata. Można skonfigurować zadanie w dwóch różnych trybach:
    a. Operacja wysyłania ma być blokująca – tzn. powrót z metody `Send()` nastąpi dopiero po faktycznym wysłaniu danych.
    b. Błąd podczas wysyłania ma powodować restart zadania.
2. `BufferSendingTask` – klasa w metodzie `InternalRun()` wywołuje metodę `Send()` z nadklasy. Klasa jest wykorzystywana do generowania zadań wysyłających proste komunikaty (np. komunikat o typie `OBTAINED_SESSION`).
3. `SessionPropagateTask` – klasa jest odpowiedzialna za wysłanie repliki konkretnej sesji do konkretnego adresata. W metodzie `InternalRun()` serializuje obiekt sesji i próbuje wysłać dane do węzła (wywołując metodę `Send()` z nadklasy). Klasa wykorzystuje tryb restartowania w przypadku błędu. W ten sposób, jeżeli z jakiegoś powodu wysłanie danych zaczyna się opóźniać, to nie blokuje niepotrzebnie pamięci zserializowanymi danymi sesji i co ważniejsze przy kolejnym uruchomieniu zadania ponawia próbę replikacji, ale już być może z nowszą wersją sesji, zmniejszając w ten sposób liczbę przesyłanych informacji (tworzenie zadań replikacyjnych zostanie opisane w p. 4.3.5). Ponadto, jeżeli system wykorzystuje buforowanie, to przy definicji zadania ustala się opóźnienie w wykonaniu (por. p. 4.3.4).
4. `ReleaseSessionTask` – zadania tej klasy są odpowiedzialne za zwalnianie sesji w serwerze zarządcy. System podczas tworzenia ustawia zależność tego zadania od wszystkich zadań replikacyjnych sesji. Proces zwolnienia jest uruchamiany dopiero po poprawnym przesłaniu replik do wszystkich węzłów w klastrze. W przypadku wykorzystywania buforowania system dodatkowo opóźnia start zadania już po wykonaniu wszystkich replikacji (szerzej na ten temat będzie mowa w p. 4.3.4).
5. `MessageReceivedTask` – zadanie tej klasy tworzone jest po odebraniu wiadomości z sieci. Zadanie zawiera dane komunikatu oraz adres węzła, który je przysłał. Przetwarzanie wiadomości wykonywane jest w odrębnym wątku, co powoduje, że nie następuje blokowanie wątku odbierającego dane z sieci. W ten sposób zapewniane jest maksymalne zrównoleglenie obliczeń wykonywanych przez moduł klastra, co może nie być bez znaczenia w przypadku serwerów wieloprocesorowych.
6. `SessionReceivedTask` – klasa dziedziczy po `MessageReceivedTask` dodając w metodzie `internalRun()` obsługę komunikatów charakterystycznych w sytuacji, gdy moduł jest zanurzony wewnątrz serwera Tomcat, jak np. komunikat `SESSIONS_ADD`.

W systemie zostały stworzone dwie odrębne kolejki zadań:

- dla zadań przetwarzających odebrane z sieci komunikaty oraz
- dla pozostałych zadań występujących w systemie (czyli w przeważającej liczbie zadań wysyłających komunikaty).

Potrzeba odizolowania tych zadań wynikła z faktu, że zadania przetwarzania wiadomości odebranych nie mogą być blokowane przez zadania wysyłające dane.

Można sobie wyobrazić sytuację, gdy wszystkie wątki w kolejce zostały wstrzymane przy próbie wysłania danych i nie ma wolnego wątka, który opróżniłby bufory z odebranymi wiadomościami. Bardzo często to właśnie szybkie przetworzenie odebranych informacji będzie miało krytyczny wpływ na wydajność systemu. W pakietach odbieranych będą informacje o przydzieleniu sesji (komunikat `OBTAINED_SESSION`) – zinterpretowanie tego komunikatu powoduje bezpośrednie wznowienie wykonywania zgłoszenia wygenerowanego przez użytkownika systemu.

Moduł klastra umożliwia konfigurowanie liczby wątków dołączonych do każdej z kolejek (opis konfiguracji znajduje się w p. 4.4).

4.3.3.1 Implementacja kolejki zadań

Do implementacji kolejki zostało wykorzystane drzewo (standardowa klasa Javy `java.util.TreeSet`) z wartościami posortowanymi zgodnie z czasem wykonania. Zadania, które mają być wykonane bez opóźnienia wstawiane są do drzewa z małymi wartościami, natomiast te, które mają się wykonać po upływie pewnego czasu są wstawiane z liczbą milisekund określającą czas wykonania.

Każdy wątek podłączony do drzewa w nieskończonej pętli pobiera zadanie z kolejki (`WorkerQueue.GetTask()` – wywołanie metody jest blokujące i w przypadku braku zadań powoduje wstrzymanie wykonywania). Następnie dla pobranego zadania wywołuje metodę `Run()`, która obudowuje wywołanie `InternalRun()`.

Głównym powodem stworzenia kolejki zadań zależnych była potrzeba realizacji procesu zwalniania sesji dopiero po udanym zakończeniu rozsyłania kopii do wszystkich węzłów klastra. Drugim, nie mniej ważnym powodem było umożliwienie realizacji pewnych zadań z opóźnieniem, jak np. wysłanie repliki po pewnym czasie, aby ewentualnie umożliwić użytkownikowi wprowadzenie kolejnych zmian i ominąć wysłanie wcześniejszej wersji.

Skuteczność i wydajność systemów rozproszonych w bardzo dużym stopniu zależy od zastosowanych metod buforowania. Wszelkiego rodzaju opóźnianie wysłania danych w celu ich zagregowania czy eliminowanie przesyłania niepotrzebnych wiadomości powodują, że system może w znacznym stopniu przyspieszyć swoje działanie. Stworzony moduł klastra również zawiera mechanizmy usprawniające jego działanie.

4.3.4 Buforowanie

W systemie zostało zastosowane buforowanie w dwóch różnych etapach:

1. Opóźnianie momentu replikacji sesji.
2. Opóźnianie momentu zwalniania sesji.

Oba typy buforowania są możliwe do zastosowania tylko w przypadku wykorzystania zintegrowanego serwera ruchu. Jeżeli serwer rozdzielający zadania na poszczególne węzły nie będzie posiadał informacji o węźle aktualnie zajmującym sesję, to buforowanie na poziomie sesji może jedynie niepotrzebnie opóźniać działanie klastra. Jeżeli natomiast posiada takie informacje, to będzie przekierowywał żądania do węzła, który aktualnie okupuje sesję, co umożliwia węzłom opóźnianie momentu przekazania sesji.

Opóźnianie momentu replikacji sesji

Każde zadanie replikujące posiada numer sesji oraz adres węzła, do którego ma wysłać replikę. Moduł dla każdego węzła przechowuje informację o stworzonych, ale nie rozpoczętych zadaniach replikujących, z możliwością wyszukiwania po identyfikatorze sesji (słownik, w którym kluczami są identyfikatory). W momencie zakończenia przetwarzania żądania, system tworząc zadania replikujące sprawdza czy nie istnieje już identyczne zadanie, które jeszcze nie zdążyło się wykonać. Jeżeli istnieje, to nie ma potrzeby tworzenia kolejnego zadania, bo istniejące tak czy owak w momencie wykonania pobierze najświeższe dane sesji. Należy tylko do zbioru słuchaczy istniejącego zadania dodać obiekt klasy `ReleaseSessionTask`, aby w odpowiednim momencie zwolnić sesję.

Takie podejście gwarantuje, że nawet jeżeli wiele wątków jednocześnie będzie przetwarzało żądania odwołujące się do tej samej sesji, to tak czy owak zostanie stworzone tylko jedno zadanie replikujące, zmniejszając w ten sposób liczbę przesyłanych informacji. Dodatkowo mechanizm pozwala na łatwą realizację koncepcji opóźniania procesu replikacji. Wystarczy, że podczas tworzenia zadania replikującego moduł wstawi opóźnienie w jego wykonanie. Jeżeli w czasie oczekiwania zostanie przetworzone kolejne żądanie (zmieniające dane sesji), to moduł ominie wysłanie wcześniejszej wersji. W ten sposób doprowadza się do sytuacji, gdzie wydajność klastra przestaje zależeć od częstotliwości zgłaszania się użytkowników. W rezultacie sesja będzie replikowane co pewien czas, niezależnie od tego czy użytkownik zgłosi się do systemu raz czy bardzo wiele razy. Dobór długości opóźnienia nie jest rzeczą zupełnie trywialną – jeżeli będzie za małe, to buforowanie może nie przynosić oczekiwanych rezultatów. Z kolei jeżeli opóźnienie będzie za duże, to system straci swoją główną zaletę – czyli dynamiczne równoważenie obciążenia. Doprowadzi to do sytuacji, gdzie każda sesja jest na stałe przypisana do konkretnego węzła – czyli analogicznie do koncepcji zastosowanej w serwerze Bea Weblogic 8.0 (por. p. 2.3.2.1), a cały nakład związany z synchronizowaniem sesji okaże się zbędny.

Opóźnianie wysłania zmienionych danych zwiększa niebezpieczeństwo utraty zmian. Jeżeli nastąpi awaria węzła, to może to doprowadzić do utraty nie tylko właśnie przetwarzanych sesji, ale również tych, które były przetworzone wcześniej, ale oczekiwały na moment replikacji. Problem ten został rozwiązany przez tworzenie jednego zadania replikacyjnego bez opóźnienia. To znaczy moduł po przetworzeniu żądania wybiera losowo jeden węzeł, dla którego tworzy zadanie wysłania repliki z natychmiastowym czasem wykonania. W ten sposób w każdej chwili najświeższe dane sesji znajdują się przynajmniej na dwóch maszynach. W razie awarii dane sesji zostaną odzyskane z drugiej maszyny (patrz opis komunikatów `CHECK_SESSION_VERSION` oraz `SESSION_VERSION` w p. 4.3.2).

Opóźnianie momentu zwalniania sesji

Poza opóźnianiem wysłania pakietów z kopią najnowszej wersji sesji można również opóźniać moment zwolnienia sesji. Wydajność klastra jest mierzona szybkością reakcji systemu na zgłoszenie użytkownika. Im reakcja jest szybsza, tym lepiej. Niestety architektura zaimplementowanego klastra wymaga, aby przed modyfikacją danych użytkownika system zablokował sesję w serwerze zarządzającym. Operacja ta jest dosyć kosztowna – wymaga przesłania dwóch pakietów w sieci (prośba o zablokowanie i odpowiedź). Jeżeli węzeł opóźni moment oddania sesji, to może przetworzyć kilka kolejnych żądań bez potrzeby ponownego

zajmowania sesji. Oczywiście nie można przesadzić z długością opóźnienia, aby nie doprowadzić do sytuacji opisanej w poprzednim punkcie, gdy klaster zaczyna się zachowywać tak, jakby istniały stałe przypisania sesja – węzeł macierzysty.

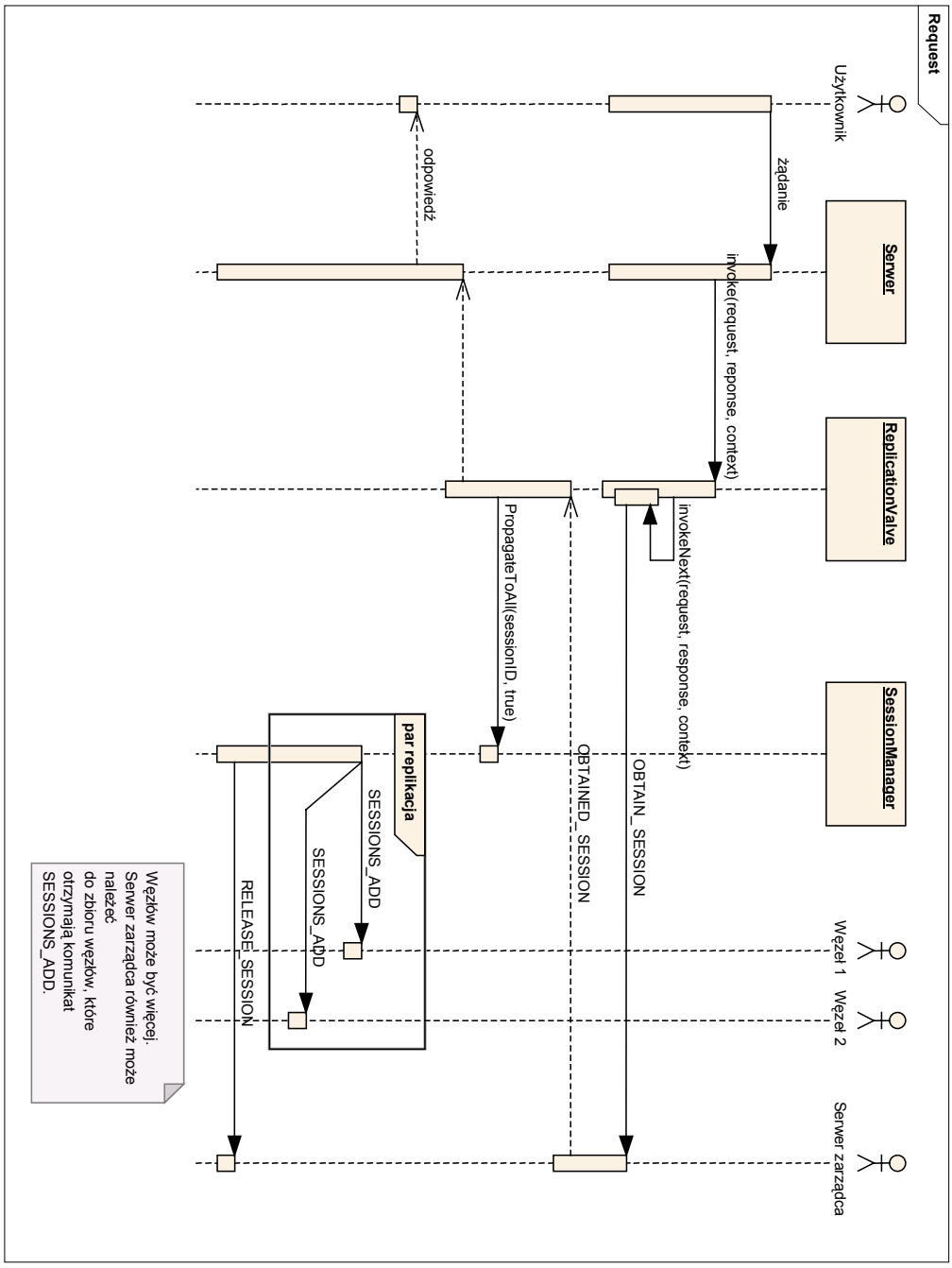
Warto zwrócić uwagę na fakt, że blokada jest przyznawana całemu węzłowi i wszystkie wątki wewnątrz serwera mogą korzystać z danych sesji bez potrzeby ponownego wysyłania prośby o jej przyznanie. W danym momencie tylko jeden wątek wysyła komunikat z prośbą o zablokowanie sesji. Wszystkie kolejne wątki, odwołujące się do sesji będą czekały na zakończenie wcześniej rozpoczętej procedury. Podobnie wygląda proces zwalniania – dopiero ostatni zwalniający wątek wyśle faktycznie komunikat do serwera zarządcy. Wszystkie wcześniejsze po prostu zmniejszą licznik odwołań.

Znając podstawowe mechanizmy działające w module można przeanalizować algorytm przetwarzania żądania. Ścieżka wykonania algorytmu rozpoczyna się w momencie zgłoszenia przez użytkownika żądania, a kończy w momencie wysłania do serwera zarządcy wiadomości zwalniającej sesję.

4.3.5 Algorytm przetwarzania żądania

Opisywana ścieżka przetwarzania żądania rozpoczyna się już w konkretnym węźle, czyli serwerze Tomcat. Sposób działania systemu na poziomie zarządcy ruchu zostanie opisany w p. 4.3.7. Koniec opisywanego procesu wyznaczany jest przez wysłanie danych z serwera Tomcat oraz przez zwolnienie blokady sesji użytkownika.

Na rys. 4.11 znajduje się diagram przepływów ukazujący najbardziej charakterystyczny przypadek obsługi żądania. Diagram przedstawia zgłoszenie klienta z numerem sesji, którą należy zablokować w serwerze zarządzającym.



Rys. 4.11. Diagram przepływu podczas obsługi żądania

Klient generuje zgłoszenie, które zostaje wysłane do serwera Tomcat. Serwer inicjuje wykonanie strumienia przetwarzania żądań (por. p. 3.3.2). Wśród załączonych wyzwalaczy występuje również wyzwalacz klasy `ReplicationValve` (analogicznie do implementacji Filipa Hanika). W metodzie `invoke(Request, Response, Context)` wyzwalacz wznawia przetwarzanie strumienia w celu wykonania zgłoszenia. Po powrocie z metody `invokeNext(Request, Response, Context)` program sprawdza czy żądanie posiada sesję oraz czy sesja jest dzielona przez cały klaster. Jeżeli tak, to zwiększa wersję sesji, tworzy dla każdego węzła w klastrze zadanie replikacyjne oraz tworzy zadanie odblokowania sesji, które wykona się po rozesłaniu wszystkich kopii. Następnie kończy działanie, tym samym kończąc

proces przetwarzania żądania. Wszystkie stworzone zadania wykonane zostaną asynchronicznie przez wątki obsługujące kolejkę zadań.

Oczywiście zadania replikacyjne zostaną stworzone tylko pod warunkiem, że w systemie nie występowały już wcześniej zdefiniowane identyczne zadania. Po wykonaniu replikatorów (zadań replikacyjnych) do kolejki trafia zadanie odblokowania sesji. W czasie uruchomienia sprawdza czy inne wątki aktualnie nie używają sesji. Jeżeli nie, to ustawia w strukturach danych, że sesja nie jest zablokowana i wysyła komunikat do serwera zarządcy w celu faktycznego odblokowania. Od tego momentu każdy wątek, który spróbuje się odwołać do sesji będzie musiał najpierw zablokować ją u zarządcy. Tutaj warto zwrócić uwagę na implementację procesu przydzielania i zwalniania zasobu. Nawet jeżeli zostaną wysłane równolegle dwa pakiety: jeden z prośbą o zwolnienie, a drugi o zablokowanie sesji, to ponieważ serwer zlicza ile razy semafor został podniesiony i opuszcza go dokładnie tyle razy ile wysłany pakiet to specyfikuje, nie będzie niebezpieczeństwa wejścia do sekcji krytycznej bez podniesienia semafora. Po prostu jeden pakiet zmniejszy licznik, a drugi go zwiększy – kolejność nie będzie odgrywała roli.

4.3.5.1 Proces tworzenia sesji

Jeżeli wygenerowane żądanie nie posiadało wcześniej utworzonej sesji, a aplikacja odwoła się do niej, to serwer Tomcat standardowo wywoła dla menedżera sesji metodę `createSession()`. Ponieważ dla aplikacji zadeklarowanych jako rozproszone (tag `<distributable/>` w pliku web.xml) menedżerem jest obiekt klasy `org..server.SessionManager`, wywołanie to zostanie przechwycone przez moduł klastra. Menedżer utworzy sesję (analogicznie jak w zwykłych menedżerach), ale sesja będzie instancją klasy `org..session.ReplicatedSession` (podklasa `org.apache.catalina.session.StandardSession`). Dodatkowo menedżer wygeneruje zadanie wysłania wiadomości `SESSION_CREATED` do serwera zarządcy informujące o stworzeniu nowego identyfikatora (zadanie będzie wykonywane w tle). Serwer zarządca przetwarzając tę wiadomość przy okazji podniesie semafor dla nowej sesji *a konto* węzła, który ją stworzył (aby nie trzeba było wysyłać kolejnego pakietu z prośbą o przyznanie sesji).

Oprócz wysłania wiadomości menedżer ustawi w globalnych strukturach danych, że sesja została przydzielona temu węzłowi. Na tym wywołanie metody `createSession()` się kończy. Dalej następuje przetwarzanie analogiczne do sytuacji, gdy sesja była utworzona wcześniej.

4.3.5.2 Proces sprawdzania istnienia sesji

Kolejnym przypadkiem jaki może się wydarzyć jest odwołanie do sesji, która nie jest zarejestrowana w menedżerze. Są dwie możliwości zaistnienia takiej sytuacji (nie licząc złośliwych żądań generowanych przez włamywaczy):

1. Sesja już wygasła w klastrze.
2. Węzeł nie otrzymał jeszcze pakietu `SESSIONS_ADD` od twórcy sesji.

O ile w pierwszym przypadku menedżer przez pewien czas może zachowywać informacje o wygasłych sesjach i bez potrzeby komunikowania się z zarządcą po prostu tworzyć nową sesję, o tyle w drugim przypadku konieczne jest wysłanie zapytania.

Obiekt klasy `SessionManager` w metodzie `findSession(String sessionID)` wykonuje następujący kod:

```
public Session findSession(String sSessionID)
     throws java.io.IOException {
   return findSession(sSessionID, true);
}
public Session findSession(String sSessionID, boolean bCreate)
     throws java.io.IOException {
   if (sSessionID == null)
      return null;
   Session oSession = super.findSession(sSessionID);
   if (oSession == null && sSessionID != null) {
      // sprawdzamy czy sesja istnieje w klastrze
      if (oSyncServer.CheckExistance(new SessionContext(sSessionID,
                  this.getName()))) {
            synchronized (sessions) {
            oSession = super.findSession(sSessionID);
            if (oSession == null) {
                  // sesja istnieje, ale my wciąż jej nie mamy
                  // dlatego tworzymy pustą, aby wątek mógł
                  // przy odwołaniu rozpocząć procedurę
                  // blokowania sesji.
                  oSession =
                        this.createSession(sSessionID, false);
                  }
            }
      }
   }
   return oSession;
}
```

Obiekt `oSyncServer` jest instancją klasy `org..server.SyncManager` (klasa została szerzej opisana w p. 4.3.6). Każdy węzeł klastra zawiera dokładnie jeden obiekt tej klasy, współdzielony przez wszystkie menedżery sesji. Metoda `CheckExistance(SessionContext)` działa analogicznie do procedury zajmowania sesji. To znaczy pierwszy wątek inicjuje faktyczną procedurę sprawdzania u serwera zarządcy (wysyła komunikat `CHECK_SESSION_EXISTANCE`), a wszystkie kolejne jedynie czekają na odpowiedź. Jeżeli okaże się, że w klastrze istnieje sesja o podanym identyfikatorze, to menedżer stworzy pusty obiekt i pozwoli na dalsze wykonywanie wątku. Nie istnieje tu niebezpieczeństwo rozspójnienia, ponieważ mimo stworzenia pustej sesji nie została ustawiona na niej blokada. Czyli wątek, który będzie się odwoływał do danych sesji, tak czy owak będzie musiał ją najpierw zablokować. Z kolei, aby blokada się udała musi zakończyć się proces replikacji, który spowoduje, że pusty do tej pory obiekt sesji zostanie w końcu zasilony danymi.

Należy tu zwrócić uwagę na fakt, że w przypadku wykorzystania zintegrowanego serwera ruchu węzły nie będą miały potrzeby generowania dodatkowych zapytań do serwera zarządcy. Jeżeli jakaś sesja będzie w danym momencie zajęta, to żądanie będzie przekierowane do zajmującego ją węzła. Jeżeli nie będzie przez nikogo zajęta, to będzie to oznaczało, że tak czy owak wszystkie węzły posiadają kopię tej sesji i nie będą musiały się pytać serwera czy istnieje.

4.3.5.3 Proces blokowania sesji poprzez odwołanie

W celu zminimalizowania niepotrzebnych operacji blokowania oraz replikowania sesji moment wejścia do sekcji krytycznej został przeniesiony w miejsce faktycznego odwołania do danych sesji. Czyli jeżeli użytkownik wygeneruje żądanie, które nie będzie zaglądało do danych sesji (nie zostaną wywołane na sesji metody `getAttribute(String)`, `setAttribute(String, Object)` itp.), to nie spowoduje to żadnego ruchu po stronie klastra.

W dalszej części tego punktu opisano sposób zaimplementowania tego mechanizmu.

Obiekt klasy `ReplicatedSession` nadpisuje wywołania wszystkich metod związanych z pobieraniem lub ustawianiem danych w sesji (czyli m.in. `getAttribute(String)`, `setAttribute(String, Object)`). W każdej z tych metod zanim zostanie wykonana metoda z nadklasy sprawdzane jest czy odwołujący się wątek zablokował już tę sesję. Jeżeli nie, to wywoływana jest metoda `ObtainSession(String sessionID)` na menedżerze sesji. To wywołanie z kolei blokuje sesję u serwera zarządcy (synchronicznie) lub zwiększa licznik odwołań do już zajętej sesji. Po przetworzeniu żądania (w wyzwalaczu `ReplicationValve`) sprawdzane jest czy wątek zajmował sesję. Jeżeli tak, to inicjowana jest replikacja oraz zwolnienie sesji.

Niestety istnieje tu hipotetyczne niebezpieczeństwo, że jeżeli aplikacja przetwarzając żądanie stworzy nowy wątek, który odwoła się do sesji, to później sesja ta nie zostanie zwolniona. Wątek założy blokadę, której później nie będzie w stanie zdjąć. Jednak sytuacja taka jest na tyle specyficzna, że raczej istnieje małe prawdopodobieństwo, że programiści zdecydują się na jej realizację w rzeczywistych aplikacjach. Nawet gdyby ktoś chciał zaimplementować taką architekturę rozwiązania, to wystarczy, że do nowego wątku przekaże już pobrane z sesji atrybuty, a po zakończeniu jego wykonywania wpisze je z powrotem.

Tak czy owak w najgorszym wypadku węzeł po prostu nie zwolni sesji, co przy wykorzystaniu zintegrowanego serwera ruchu nie spowoduje zaprzestania działania aplikacji. Wszystkie żądania będą kierowane na jedną maszynę bez możliwości zmiany kontekstu wykonania. Atutem zastosowania klastra będzie natomiast wciąż trwający proces replikowania sesji, co może okazać się nie bez znaczenia w przypadku awarii tego węzła.

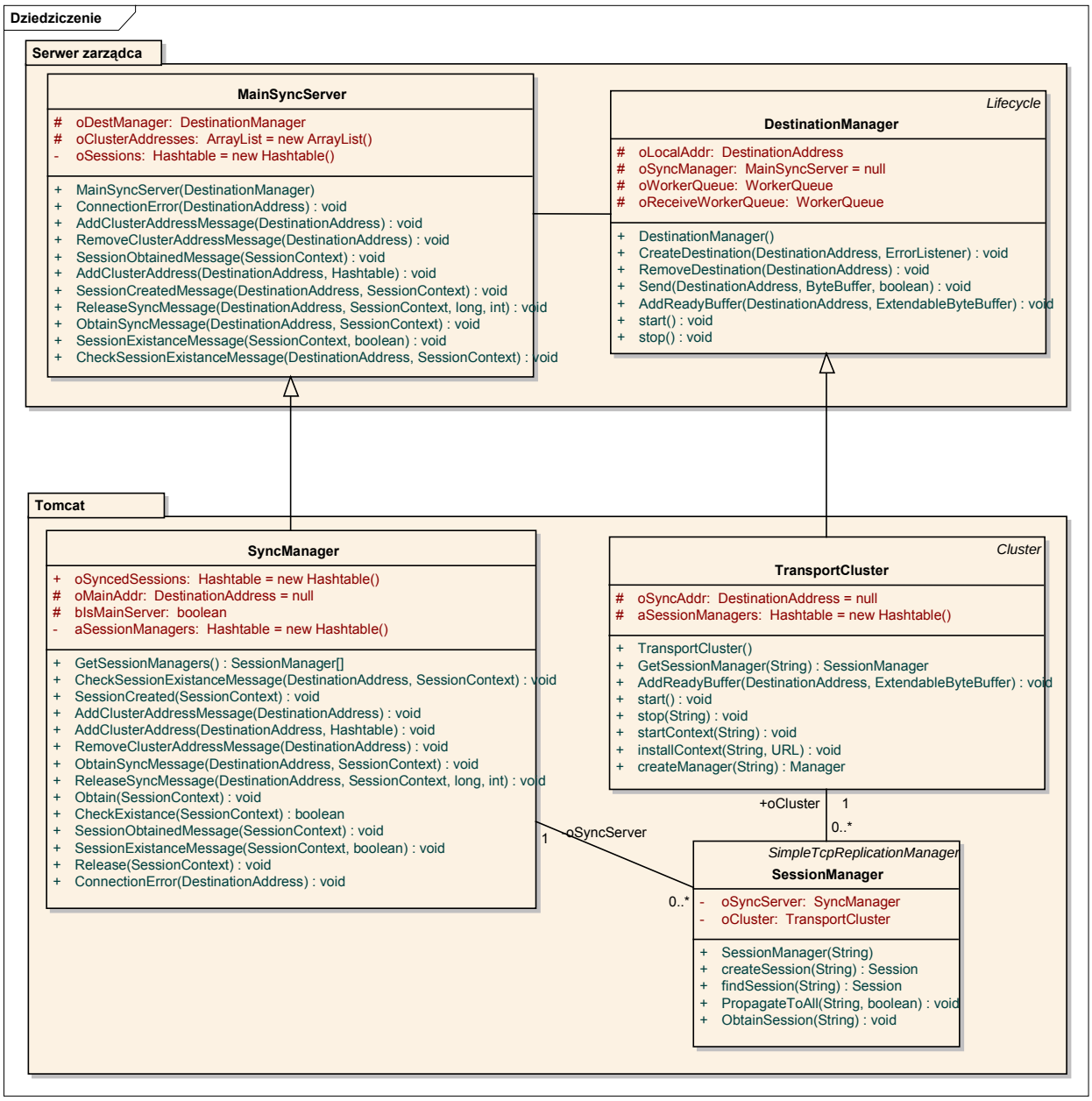
Cała logika związana z blokowaniem, zwalnianiem czy replikowaniem sesji znajduje się w klasie menedżera sesji (`org..server.SessionManager`) oraz klasie samej sesji (`org..session.ReplicatedSession`). Niemniej jednak większość kodu niezbędnego do komunikacji z serwerem zarządcą jak i kod samego serwera zarządcy znajduje się w dwóch klasach: `org..server.MainSyncServer` oraz `org..server.SyncManager`.

4.3.6 Serwer zarządca

Głównym celem implementacji serwera zarządcy była możliwość wykorzystania tego samego kodu w dwóch przypadkach:

1. Serwer zarządca jest jednym z węzłów klastra (zamieszczony wewnątrz Tomcata).
2. Serwer zarządca jest osobnym programem, który nie ma nic wspólnego z klasami Tomcata.

Aby uzyskać taką dwoistość modułu, zastosowano mechanizm dziedziczenia w podejściu obiektowym. Implementacja bazowa serwera zarządcy opiera się na klasie `org..server.MainSyncServer` oraz klasie implementującej warstwę transportującą – `org..server.DestinationManager`. Zastosowanie modułu wewnątrz serwera Tomcat staje się możliwe poprzez dodanie klas dziedziczących po klasach bazowych (por. rys. 4.12).



Rys. 4.12. Hierarchia klas podzielona na dwa możliwe zastosowania

W wywołaniach nadpisanych metod został dodany kod specyficzny dla serwera Tomcat. W szczególności został zaimplementowany mechanizm zdalnego lub lokalnego (w zależności od konfiguracji) odwoływania się do serwera zarządcy. Jeżeli moduł ma działać jako klient zdalnego serwera zarządcy, to wszystkie odwołania do

metod z klasy `SyncManager` powodują wygenerowanie odpowiednich komunikatów w sieci. Natomiast w przypadku, gdy moduł wewnątrz Tomcata pełni jednocześnie rolę serwera zarządcy, wtedy klasa `SyncManager` wywołuje kod z nadklasy, czyli `MainSyncServer`. Czyli tak naprawdę zainstalowanie pojedynczego serwera w klastrze (w trybie serwera zarządcy) nie spowoduje znaczącego spadku wydajności w stosunku do sytuacji, gdy serwer ten będzie działał w ogóle bez modułu klastra. Taka informacja może mieć duży wpływ na podjęcie decyzji o instalacji środowiska rozproszonego. Administrator instalując klaster nie jest zmuszony do natychmiastowego podłączenia wielu komputerów, w celu zrekompensowania spadku mocy obliczeniowej. Pojedyncza maszyna będzie działała równie wydajnie jak przed podłączeniem modułu.

Wydajność klastra w dużej mierze zależy od oprogramowania rozdzielającego zadania na poszczególne jego węzły. Szczególnie istotne jest to w przypadku przedstawionego w pracy rozwiązania. Algorytm, który nie wykorzystuje informacji o aktualnych przydziałach sesji, spowoduje osłabienie mocy przerobowej systemu i uniemożliwi wykorzystanie buforowania.

4.3.7 Serwer ruchu

Zasadniczym celem pracy było napisanie i przedstawienie działającego modułu klastra w serwerze Tomcat. Niemniej jednak, aby móc w całości ukazać działanie rozwiązania konieczne stało się napisanie zintegrowanego serwera przekierowującego żądania do poszczególnych maszyn klastra. Niestety nie udało się znaleźć gotowego programu, który odznaczałby się następującymi cechami:

- Napisany w języku Java z dostępnym kodem źródłowym.
- Bardzo wydajny (przekierowania na poziomie TCP/IP).
- Buforujący pulę połączeń.

Dlatego w ramach pracy został napisany serwer ruchu jednocześnie działający jako serwer zarządca. Implementację należy traktować jako wzorcowy przykład rozwiązania, a nie jako docelowy i w pełni działający serwer do przekierowywania żądań.

Serwer został w całości napisany w języku Java z wykorzystaniem klas bazowych z modułu klastra. Schemat działania programu jest stosunkowo prosty:

1. Przy starcie otwiera port, na którym będzie przyjmował żądania od klientów.
2. Tworzy i inicjuje obiekty serwera zarządcy.
3. W momencie, gdy jakiś węzeł klastra zgłosi się do serwera zarządcy jednocześnie dodawany jest do listy aktywnych serwerów Tomcat.
4. Przy zgłoszeniu klienta program wybiera serwer Tomcat z listy aktywnych i zaczyna się zachowywać jak serwer pośredniczący, przekazując dane między klientem a serwerem.
5. Jeżeli któraś ze stron zakończy połączenie, to program automatycznie kończy połączenie z drugiej strony.
6. W momencie zgłoszenia awarii węzła przez oprogramowanie serwera zarządcy jest on automatycznie usuwany z listy aktywnych serwerów Tomcat.

4.3.7.1 Szczegóły implementacyjne

Dla każdego aktywnego serwera Tomcat trzymana jest pula połączeń (jej wielkość jest konfigurowalna). Każde połączenie jest zadaniem, które w momencie

uruchomienia powoduje rozpoczęcie czytania z kanału komunikacyjnego. Jednocześnie obiekt zadania posiada metodę `Write(ByteBuffer)`, która umożliwia pisanie danych do połączenia. Przed ponownym uruchomieniem zadania (wywołaniem metody `Restart()`) ustawiane są referencje między zadaniem obsługującym połączenie z serwerem, a zadaniem obsługującym połączenie z klientem. W ten sposób później wszystkie odebrane informacje przez jeden obiekt zostają przesłane do drugiego (tworząc pomost dla danych). Różnice między zadaniami klienta a serwera występują w zachowaniu na początku i końcu.

Zadanie połączenia klienta na początku, zanim zostanie sparowane z zadaniem serwerowym, czyta nagłówek żądania w celu sprawdzenia czy nie odwołuje się ono do konkretnej sesji. Jeżeli tak, to program sprawdza czy podana sesja nie jest aktualnie zajmowana przez jeden z węzłów. Jeżeli jest zajmowana, to zadanie jako odbiorcę wybiera połączenie z puli tego węzła. W przeciwnym przypadku zadanie wybiera serwer posiadający najwięcej nieużywanych połączeń.

Z kolei zadanie serwerowe po skończeniu obsługiwania klienta odnawia połączenie z serwerem, aby przy obsłudze kolejnego żądania nie trzeba było tracić czasu na nawiązywanie połączenia. Istotne jest, aby odpowiednio skonfigurować połączenia w serwerach – tzn. dopuszczalny czas bezruchu na połączeniu musi być odpowiednio długi, aby połączenia nie były zbyt szybko zrywane przez serwer. Jeżeli połączenie zostanie zerwane zanim jakiś klient zacznie je wykorzystywać, to obsługa żądania zostanie wydłużona o czas odnowienia połączenia.

Wykorzystanie zadań umożliwiło dynamiczny przydział wątków do obsługi wszystkich połączeń ze wszystkich serwerów jednocześnie. W ten sposób można zwiększyć liczbę oczekujących połączeń bez zwiększania liczby wątków. Dynamiczny przydział wątków ze wspólnej puli powoduje, że rozwiązanie dużo lepiej adaptuje się do aktualnego zapotrzebowania na połączenia.

Warto zwrócić uwagę, że zastosowane rozwiązanie umożliwia wykorzystanie opcji protokołu HTTP 1.1, `Connection: Keep-Alive`. Opcja pozwala przeglądarce na wykorzystanie raz otwartego połączenia dla obsługi kilku osobnych żądań. Po prostu żadna ze stron nie zerwie połączenia, tym samym zachowując aktywny pomost w serwerze ruchu.

Wadą zaproponowanego rozwiązania jest brak interpretera nagłówka protokołu HTTP. Gdyby program był w stanie poprawnie interpretować nagłówek mógłby zachowywać otwarte połączenia z każdym z węzłów, bez potrzeby ich odnawiania (do tego niezbędna byłaby kontrola długości przesyłanych danych). Wtedy również w przypadku trzymania połączeń typu `Keep-Alive` można by było dynamicznie zmieniać serwer, który obsłuży kolejne żądanie. Niestety stopień skomplikowania takiego interpretera wykracza poza ramy tej pracy.

Zarówno serwer ruchu jak i sam moduł klastra zawierają szereg parametrów, które umożliwiają dostrojenie systemu w zależności od zastosowania. W kolejnym punkcie opisano parametry oraz przedstawiono różne dodatkowe wskazówki, którymi mogą się kierować administratorzy klastra.

4.4 Konfiguracja klastra

Parametry klastra ustawia się w pliku konfiguracyjnym `server.xml` (analogicznie do rozwiązania z wersji 5.0). Należy wstawić znacznik `Cluster` w pliku opisującym ustawienia serwera Tomcat:

```
<Cluster
  className="org..transport.TransportCluster"
  name="ReliableTransportCluster"
  debug="3"
  tcpListenAddr="192.168.0.2:4001"
  syncServer="192.168.0.2:4010"
  sendThreadCount="5"
  receiveThreadCount="3"
  receiveBufferCount="10"
  syncSessionHoldingTimeout="20000"
  sessionObtainTimeout="40000"
  serverSideConnTimeout="10000"
  clientSideReconnectCount="5"
  replicationBufferingTime="0"
  releaseSessionTimeout="0"
/>
```

Opis poszczególnych parametrów:

- `className` – nazwa klasy implementującej interfejs `org..Cluster`.
- `name` – nazwa klastra (tylko informacyjnie).
- `debug` – poziom szczegółowości przy wypisywaniu komunikatów przez moduł klastra (od 0 do 10, przy czym dla 10 wypisywane jest wszystko).
- `tcpListenAddr` – adres, na którym będzie nasłuchiwał moduł klastra. Jeżeli adres przed dwukropkiem nie zostanie podany, to moduł pobierze domyślny adres lokalny z maszyny (czyli np. `„:4001”`).
- `syncServer` – adres serwera zarządcy. Jeżeli adres będzie pusty, to przyjmuje się, że ten serwer ma działać w trybie serwera zarządcy.
- `sendThreadCount` – liczba wątków przetwarzających zadania wysyłające dane.
- `receiveThreadCount` – liczba wątków przetwarzających odebrane z sieci komunikaty.
- `receiveBufferCount` – liczba buforów, do których będą wpisywane komunikaty pobrane z sieci. Powinno ich być więcej niż wątków, przetwarzających odebrane komunikaty. Z drugiej strony nie może ich być za dużo, aby nadmiernie nie obciążać pamięci maszyny.
- `syncSessionHoldingTimeout` – parametr odnosi się do serwera zarządcy. Określa czas w milisekundach po jakim serwer kończy zbieranie informacji o tym, który węzeł posiada najwyższy numer wersji danej sesji (np. po awarii któregoś węzła).
- `sessionObtainTimeout` – parametr określa maksymalny czas oczekiwania na zajęcie sesji w serwerze zarządcy. Jeżeli czas zostanie przekroczony, to moduł zgłosi wyjątek.
- `serverSideConnTimeout` – parametr określa czas w milisekundach, po którym moduł uznaje połączenie za zerwane, jeżeli nie uda się go ponownie nawiązać (powinien być taki sam dla wszystkich węzłów w klastrze).
- `clientSideReconnectCount` – parametr określa liczbę prób, które ma podjąć strona aktywna połączenia w celu odbudowania kanału komunikacyjnego. Próby zostaną rozłożone równomiernie w czasie określonym parametrem `serverSideConnTimeout`.

- `replicationBufferingTime` – parametr określa opóźnienie z jakim zostaną wykonane zadania replikowania sesji. Zero oznacza brak buforowania.
- `releaseSessionTimeout` – parametr określa opóźnienie z jakim zostanie zwolniona sesja (opóźnienie będzie liczone od momentu zakończenia ostatniego zadania replikującego). Zero oznacza brak opóźnienia.

Poza strojeniem parametrów klastra istnieje jeszcze możliwość usprawnienia działania systemu poprzez odpowiednie skonfigurowanie sprzętu. Skuteczność klastra najbardziej zależy od przepustowości sieci. Jeżeli brakuje funduszy na zakup super szybkiej sieci (jak np. sieci *Myrinet*), to można zainstalować dwie równoległe sieci.

4.4.1 Koncepcja dwóch sieci

Architektura klastra pozwala na logiczny i fizyczny podział przesyłanych informacji na dwie odrębne części:

1. Dane przesyłane w ramach modułu klastra.
2. Dane przesyłane między klientem a serwerem Tomcat.

Jeżeli połączy się węzły dwoma niezależnymi sieciami, to można skonfigurować klaster tak, aby te dwie grupy danych były przesyłane niezależnymi kanałami. W ten sposób można doprowadzić do sytuacji, gdzie wydajność systemu całkowicie przestaje zależeć od wydajności modułu klastra. Nawet jeżeli sieć obsługująca dane modułu klastra zacznie się „zapychać", to jeżeli wykorzysta się zintegrowany serwer ruchu, który będzie minimalizował liczbę operacji zajmowania i zwalniania sesji, to reakcja pojedynczego węzła będzie równie szybka jak w przypadku systemu złożonego z jednego serwera.

Koncepcja jest o tyle interesująca, że koszt zainstalowania drugiej sieci jest stosunkowo niewielki w porównaniu z zyskiem jaki można w ten sposób osiągnąć. Dodatkowo zainstalowanie dwóch sieci upraszcza ich monitorowanie oraz ułatwia lepsze zabezpieczenie sieci przesyłającej repliki sesji przed potencjalnymi włamywaczami.

Każdy system WWW udostępniający swoje zasoby do Internetu powinien posiadać mechanizmy zabezpieczające dane przed podsłuchem osób trzecich. Pojedynczy serwer Tomcat posiada obsługę połączeń szyfrowanych – HTTPS. Należy się zastanowić nad możliwym wykorzystaniem łączy szyfrowanych w klastrze.

4.4.2 Możliwe konfiguracje przy wykorzystaniu HTTPS

Zasadniczo istnieją dwa możliwe rozwiązania z wykorzystaniem HTTPS:

1. Zastosowanie serwera pośredniczącego, który deszyfruje dane zanim dotrą one do klastra.
2. Deszyfrowanie danych na poziomie każdego węzła osobno.

Pierwsze rozwiązanie polega na umieszczeniu serwera z odpowiednim oprogramowaniem (np. serwer HTTP Apache) lub sprzętu deszyfrującego przed serwerem ruchu. Wtedy klaster zachowuje się tak, jakby szyfrowanie nie występowało.

W drugim rozwiązaniu zakładamy, że serwer ruchu nie będzie w stanie odszyfrować przesyłanych informacji, a tylko będzie je przesyłał do wybranego przez

siebie węzła (oczywiście wtedy wszystkie węzły będą musiały posiadać identyczny certyfikat).

Oba rozwiązania mają swoje wady i zalety. W pierwszym moc obliczeniowa związana z szyfrowaniem i deszyfrowaniem jest skupiona na pojedynczej maszynie, co może się okazać wąskim gardłem systemu. Natomiast umożliwia ono wykorzystanie zintegrowanego serwera ruchu, który będzie miał dostęp do nagłówka żądania. Z kolei w drugim moc obliczeniowa związana z szyfrowaniem jest rozkładana na wszystkie węzły w klastrze, ale serwer ruchu nie może podejrzeć nagłówków zaszyfrowanych żądań i tym samym nie może stwierdzić, który węzeł aktualnie zajmuje daną sesję.

Ogólnie jeżeli klaster w większej części ma obsługiwać połączenia szyfrowane to lepiej jest zastosować drugą koncepcję, co umożliwi rozdzielenie pracy związanej z szyfrowaniem na cały klaster. Ponieważ i tak sesja jest replikowana na wszystkie maszyny, więc nie ma problemu z wyborem serwera bez znajomości nagłówków.

Natomiast jeżeli klaster w większym stopniu ma serwować treści nie wymagające szyfrowania, to lepiej jest zastosować koncepcję z pojedynczą maszyną deszyfrującą. Umożliwi to wykorzystanie metod buforowania, które usprawnią działanie systemu.

Doskonałym dowodem na słuszność postawionej tezy są wyniki testów przedstawione w punkcie 5. W rozwiązaniach opartych na połączeniach szyfrowanych zastosowanie klastra może w sposób niebagatelny zwiększyć wydajność systemu, zastępując tym samym bardzo drogi serwer, dysponujący równoważną mocą obliczeniową.

5 Testy wydajności

W tym rozdziale zostaną przedstawione wyniki przeprowadzonych testów wydajności nowego klastra. Przy projektowaniu testów były brane pod uwagę dwa zasadnicze aspekty: poprawność oraz szybkość działania. Testy zostały wykonane w laboratorium komputerowym na Wydziale Matematyki, Informatyki i Mechaniki Uniwersytetu Warszawskiego. Dostępne były dwa typy maszyn o następującej charakterystyce:

Parametr	**Maszyna typu A**	**Maszyna typu B**
Procesor	350 MHz	700 MHz
RAM	64 MB	128 MB
Sieć	100 Mb/s	100 Mb/s
SO	Linux	Linux

Do testowania został wykorzystany specjalnie w tym celu napisany program, którego zadaniem było generowanie odpowiedniej liczby równoległych żądań do systemu, uwzględniających przesłany przez system identyfikator sesji. Aby wyniki były bardziej miarodajne, program testujący był uruchamiany równolegle na kilku maszynach (w ten sposób wyniki zostały uniezależnione od wydajności konkretnej maszyny testującej, co miało szczególne znaczenie przy testowaniu połączeń szyfrowanych).

Po stronie serwera do testów zostały wykorzystane dwa typy aplikacji. Pierwsza aplikacja (dalej nazywana aplikacją sesyjną) działała w następujący sposób:

1. Pobierała z żądania nazwę parametru oraz jego wartość.
2. Wstawiała tę parę do sesji.
3. Zwracała wszystkie pary klucz-wartość znajdujące się w sesji.

Aplikacja sesyjna została napisana głównie w celu praktycznego udowodnienia poprawności działania klastra. Program testujący generując kolejne żądania, odwołujące się do tej samej sesji, sprawdzał czy wynik żądania zgadza się z oczekiwaniami. Jeżeli w wyniku nie znajdował się jeden z dodanych przez niego parametrów, to podnoszony był wyjątek i test kończył się niepowodzeniem. Należy zwrócić uwagę na fakt, iż z wydajnościowego punktu widzenia aplikacja jest skrajnie pesymistycznym przypadkiem wykorzystania klastra. Przetworzenie żądania powoduje znikome obciążenie serwera, jednocześnie zmuszając klaster do replikowania wciąż rosnących sesji.

Druga aplikacja (dalej nazywana aplikacją XSL) działała w następujący sposób:

tego z 4. Przy małym obciążeniu systemu wąskim gardłem staje się strata związana z przesyłaniem replik w sieci. Natomiast wraz ze wzrostem liczby równoległych żądań zysk z doinstalowania kolejnego węzła staje się coraz bardziej widoczny. Jest to bardzo ważna obserwacja z punktu widzenia administratora systemu. Zwiększenie liczby węzłów w klastrze musi iść w parze ze zwiększeniem liczby użytkowników systemu; w przeciwnym przypadku dodanie węzła spowoduje spadek wydajności systemu.

Test został przeprowadzony dla aplikacji sesyjnej, aby uwidocznić zysk jaki można osiągnąć przy instalacji klastra do obsługi połączeń szyfrowanych. Jeżeli test zostanie przeprowadzony dla bardziej wymagającej aplikacji, to można oczekiwać, że wyniki będą jeszcze korzystniejsze.

6 Możliwe rozszerzenia

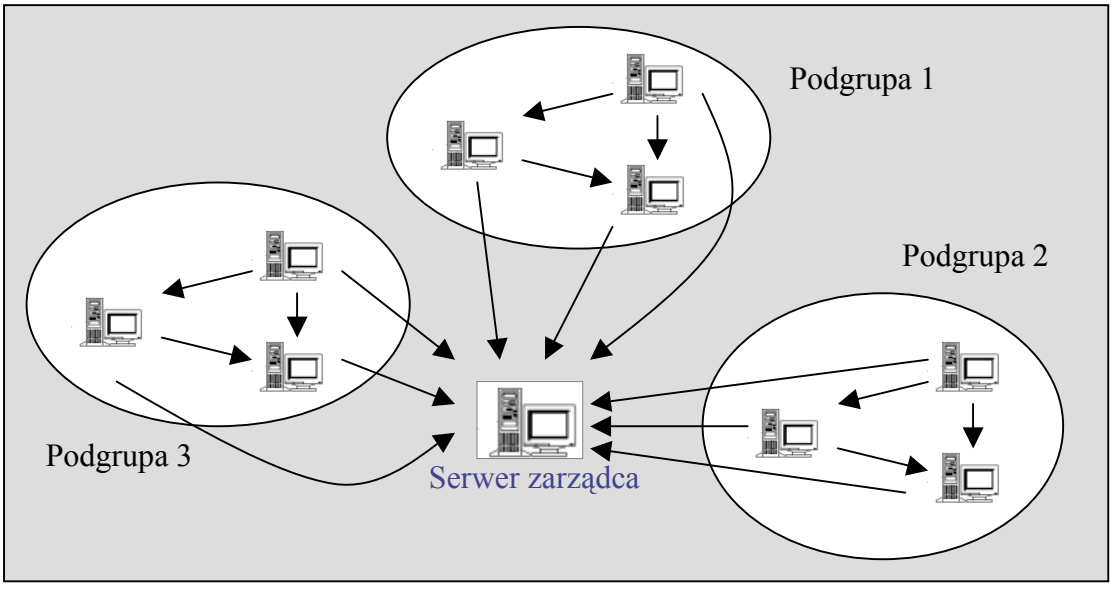
W trakcie pisania klastra pojawiło się sporo różnego rodzaju możliwych rozszerzeń, które nie zostały zaimplementowane z powodu braku czasu. W tym rozdziale zamieszczono krótki opis ciekawszych pomysłów na dalszy rozwój projektu.

6.1 Algorytm elekcji

W aktualnej wersji serwer zarządca jest ustalany na poziomie pliku konfiguracyjnego i jest na stałe zaszyty w działającym serwerze Tomcat. Ciekawym rozszerzeniem byłoby zaimplementowanie dynamicznego algorytmu elekcji, który wybierałby serwer zarządcę spośród dostępnych w klastrze węzłów. W przypadku awarii aktualnego serwera zarządcy działające komputery automatycznie ustalałyby kolejny komputer czuwający nad synchronizacją w klastrze. Taki mechanizm wyeliminowałby potencjalny słaby punkt klastra – czyli awarię centralnego serwera. W aktualnej wersji jedynym sposobem zabezpieczenia się przed awarią serwera zarządcy jest sprzętowa podmiana maszyny fizycznej na identycznie skonfigurowaną w przypadku wystąpienia awarii. Takie rozwiązanie niestety może być dosyć kosztowne.

6.2 Podział klastra na podgrupy

Sporą wadą aktualnego rozwiązania jest liniowa zależność ilości przesyłanych w sieci danych od liczby węzłów w klastrze (repliki sesji wysyłane są do każdego węzła). Klaster o bardzo dużej liczbie węzłów może okazać się nieużyteczny w praktycznych zastosowaniach. Szczególnie jeżeli sesje będą zawierały bardzo wiele danych.



Rys. 6.1. Architektura klastra z podgrupami

Istnieje możliwość zapobiegnięcia sytuacji przeładowania sieci poprzez logiczne rozbicie klastra na kilka rozłącznych podgrup. Każda podgrupa mogłaby posiadać obsługiwany przez nią podzbiór sesji – repliki byłyby rozsyłane tylko w obrębie podgrupy. Jeżeli podgrupy byłyby niedużej wielkości (np. 3, 4 węzły) ograniczyłoby

się w znacznym stopniu obciążenie sieci i przede wszystkim wyeliminowałoby się zależność liczby przesyłanych replik od liczby węzłów w klastrze. Architekturę takiego rozwiązania przedstawia rys. 6.1.

Wbrew pozorom implementacja takiego uogólnienia byłaby całkiem prosta. Każdy węzeł widzi tylko te węzły, które wskaże mu serwer zarządca. Czyli wystarczy, że serwer zarządca wskaże mu tylko węzły należące do jego podgrupy, nie informując go o istnieniu pozostałych. Dla pojedynczego węzła klaster będzie się składał tylko z węzłów z jego podgrupy. Implementacja rozszerzenia sprowadzałaby się zatem do przeprogramowania serwera zarządcy tak, aby logicznie rozbił klaster na mniejsze podgrupy. Dodatkowo konieczne byłoby zainstalowanie klastra ze zintegrowanym serwerem ruchu, aby żądania odwołujące się do sesji nie trafiły do podgrupy, która nie posiada danych sesji. Oczywiście uniemożliwiłoby to zastosowanie konfiguracji HTTPS, gdzie każdy węzeł niezależnie szyfruje i deszyfruje dane – serwer ruchu musi znać odszyfrowane nagłówki zanim przekieruje żądanie dalej (por. p. 4.4.2).

6.3 Zmniejszanie ilości przesyłanych informacji

Istnieje jeszcze drugie, równoległe podejście mające na celu zmniejszenie liczby przesyłanych w sieci bajtów. Można by było w przypadku bardzo dużych sesji wysyłać przyrosty danych. W momencie rozpoczęcia przetwarzania żądania system zapamiętywałby zserializowane dane sesji. Następnie po przetworzeniu żądania serializowałby sesję ponownie i sprawdzał co tak naprawdę się zmieniło od poprzedniej wersji. Faktyczne zmiany na poziomie konkretnych bajtów rozsyłane by były do pozostałych węzłów w sieci. Takie rozwiązanie ma dwie zasadnicze zalety:

1. Jeżeli sesja była pobrana tylko do odczytu, to system nie wygeneruje żadnych pakietów replikacyjnych.
2. Jeżeli zaszły niewielkie zmiany w sesji, to pakiety replikacyjne mogą okazać się znacznie mniejsze niż w przypadku pełnej replikacji.

Niestety wadą takiego podejścia jest łatwość utraty danych. Wystarczy, że informacje o jednej ze zmian zanikną w którymś węźle, by przy odbiorze kolejnej zmiany nie był on w stanie odtworzyć aktualnej wersji sesji. Poza tym rozwiązanie wymaga dużego, dodatkowego nakładu pamięci – trzeba przechować kopię sesji na czas przetwarzania żądania.

6.4 Implementacja interfejsu użytkownika do zarządzania klastrem

Poza rozwojem mającym na celu usprawnienie klastra warto również rozważyć rozwój związany z wygodą jego administracji. Niestety w aktualnie zaimplementowanej wersji w celu zainstalowania lub podmiany aplikacji działającej w klastrze administrator musi wykonać instalację na każdym węźle osobno[2]. Przy klastrach składających się z wielu węzłów może to być bardzo uciążliwe. Warto by było zaimplementować mechanizm automatycznej instalacji aplikacji w całym klastrze. Można by było to zrealizować na poziomie serwera zarządcy. W momencie

[2] Ewentualnie w aktualnej wersji można zainstalować każdy z serwerów na rozproszonym systemie plików (np. NFS). W momencie zmiany aplikacji wszystkie węzły od razu widziałyby nową wersję. Niemniej jednak administrator musiałby ręcznie powiadomić każdy z węzłów o konieczności przeinstalowania aplikacji.

dołączania węzła do klastra serwer zarządca mógłby wysyłać listę aplikacji (wraz z ich ścieżkami) do nowej maszyny w celu ich zainstalowania. Węzeł będąc w klastrze nie instalowałby aplikacji z lokalnego systemu plików (jak to się dzieje w standardowym serwerze), tylko pobierałby listę aplikacji podczas pierwszego zgłoszenia do zarządcy. W przypadku instalacji nowej aplikacji administrator wysyłałby odpowiednie polecenia do serwera zarządcy, który dalej rozpropagowywałby tę informację do wszystkich węzłów.

Poza tym wskazane byłoby zaimplementowanie wygodnego interfejsu użytkownika (na wzór interfejsu do administracji pojedynczym serwerem Tomcat), w którym administrator mógłby instalować, wstrzymywać czy deinstalować aplikacje. Dodatkowo miałby możliwość zmiany parametrów wszystkich węzłów w klastrze za jednym razem.

6.5 Rozbudowa serwera ruchu

Poza rozbudową modułu klastra możliwa jest również rozbudowa serwera ruchu, który na tę chwilę należy raczej traktować jako implementację wzorcową. W pełni funkcjonalna implementacja serwera ruchu została wcześniej opisana w p. 4.3.7. Należałoby dodać do serwera interpreter nagłówków HTTP, który pozwoliłby na bardzo długie przetrzymywanie otwartych połączeń z serwerami Tomcat, a przede wszystkim pozwoliłby na dynamiczne przełączanie kontekstu wykonania nawet w przypadku połączeń typu `Keep-Alive`. Mogłoby to spowodować znaczne polepszenie wydajności – szczególnie przy nawiązywaniu połączeń.

Innym bardzo praktycznym rozszerzeniem mogłoby się okazać zaimplementowanie obsługi protokołu HTTPS na poziomie serwera ruchu. Serwer samodzielnie zajmowałby się szyfrowaniem i deszyfrowaniem połączeń co wyeliminowałoby potrzebę instalacji dodatkowych komponentów (serwera pośredniczącego) w przypadku korzystania z protokołu szyfrowanego.

7 Podsumowanie

Cel, który postawiono w ramach pracy został osiągnięty. Stworzone oprogramowanie działa bardzo dobrze nawet w skrajnie trudnych warunkach. Klaster zachowuje się stabilnie, potrafiąc odpowiednio reagować na awarie węzłów. Z testów wydajności wynika, że w praktycznych zastosowaniach narzut związany z replikacją zostaje całkowicie zrekompensowany zyskiem mocy obliczeniowej złączonych maszyn. W tym momencie najsłabszym ogniwem zaproponowanego rozwiązania jest zintegrowany serwer ruchu. Aby w pełni wykorzystać możliwości modułu, powinno się usprawnić działanie tego serwera.

Z obserwacji wyników testów nasuwa się wniosek, że najmocniejszą stroną nowego klastra jest możliwość rozproszonego przetwarzania połączeń szyfrowanych. Replikacja każdy do każdego umożliwia przekierowywanie żądań do węzłów klastra bez znajomości nagłówków (dokładniej identyfikatora sesji). Takiego rozwiązania nie ma w swojej ofercie żaden komercyjny serwer na rynku. Jeżeli administrator zdecyduje się na zastosowanie połączeń szyfrowanych, to przy dużej liczbie użytkowników musi albo kupić bardzo wydajny, wieloprocesorowy serwer, albo specjalny sprzęt deszyfrujący. Oba rozwiązania są niestety bardzo kosztowne. Nowy klaster Tomcata pozwala na wykorzystanie mocy obliczeniowej wielu zwykłych komputerów do kosztownych operacji szyfrowania, jednocześnie gwarantując niezawodność replikacji i poprawne działanie w przypadku równoległych odwołań do tej samej sesji. Stworzenie klastra złożonego z kilku czy kilkunastu niedrogich komputerów stacjonarnych będzie rozwiązaniem wielokrotnie tańszym, a przy okazji dużo bardziej skalowalnym. Przy małym obciążeniu systemu można odłączyć część węzłów; jeżeli obciążenie wzrośnie można dokupić kolejną maszynę i odpowiednio skonfigurowaną wpiąć do sieci, aby odciążyła pozostałe węzły. Jeżeli okaże się, że wąskim gardłem systemu staje się serwer zarządca, to można go odseparować od serwera Tomcat i uruchomić na oddzielnej maszynie jako samodzielny program. Pewnym mankamentem klastra może być brak wygodnego narzędzia pozwalającego na administrowanie nim. Niemniej jednak, jeżeli skorzysta się z sieciowego systemu plików (NFS), to dodatkowy nakład pracy związany z utrzymaniem klastra nie musi być aż tak dotkliwy.

Sądzę, że nowy klaster może być znakomitą alternatywą dla dużych komercyjnych serwerów. Idealnie pasuje do koncepcji darmowego oprogramowania promującego tanie, ale funkcjonalne rozwiązania, które dowodzą, że w informatyce nie trzeba posiadać dużych funduszy, aby zrealizować wielkie zamiary.

Dodatek A
Opis załączników

Do pracy dołączona została płyta CD z kodami źródłowymi zaimplementowanego modułu klastra. Poniżej znajduje się hierarchia katalogów wraz z opisem zawartości.

- Dokumentacja – praca magisterska wraz z załącznikiem.
- Kody zrodlowe – kody zródłowe modułu klastra.
    - Cluster – kody modułu dołączanego do serwera Tomcat. Tutaj znajduje się również plik Build.xml umożliwiający zbudowanie modułu przy pomocy narzędzia ANT.
        - conf – plik konfiguracyjny Tomcata: server.xml.
        - dist – zbudowany moduł klastra.
    - LoadBalancer:
        - cluster – kody źródłowe serwera ruchu.
        - test – kody źródłowe aplikacji testujących.
    - Session – kod źródłowy serwletu używanego podczas testów.
    - WebXsl – kod aplikacji XSL używanej podczas testów.

Moduł klastra pisany był przy użyciu narzędzia Eclipse. Wraz z kodami źródłowymi na płycie znajdują się pliki tworzone przez to narzędzie, co umożliwia otwarcie projektów bezpośrednio w środowisku Eclipse bez potrzeby ich importowania.

Instalacja modułu klastra

Aby zainstalować moduł klastra należy:

1. zainstalować serwer Tomcat w wersji 5.0,
2. przegrać plik `/kod zrodlowy/Cluster/dist/catalina-cluster.jar` do katalogu `[Tomcat]/server/lib`, gdzie `[Tomcat]` jest katalogiem domowym zainstalowanego serwera,
3. przegrać plik konfiguracyjny `/kod zrodlowy/Cluster/conf/server.xml` do katalogu `[Tomcat]/conf`.

Aby klaster mógł działać należy jeden z węzłów skonfigurować w trybie serwera zarządcy, a w pozostałych ustawić adres serwera zarządcy (dokładniejszy opis konfiguracji znajduje się w rozdz. 4.4).

Bibliografia

[1] Abraham Kang. *J2EE Clustering, Part 1*. http://www.javaworld.com

[2] G. F. Pfister. *In Search of Clusters*. Prentice Hall, Upper Saddle River, 1998.

[3] Specyfikacja i opis standardu J2EE, http://java.sun.com/j2ee

[4] Specyfikacja Java Mail API, http://java.sun.com/products/javamail

[5] Strona domowa aplikacji Hyperion Analyzer, http://www.hyperion.com

[6] Strona domowa biblioteki do komunikacji współbieżnej Java Groups, http://www.jgroups.org

[7] Strona domowa darmowego serwera J2EE, Jonas, http://jonas.objectweb.org

[8] Strona domowa serwera Apache Tomcat, http://jakarta.apache.org/tomcat

[9] Strona domowa serwera J2EE firmy Sybase, http://www.sybase.com

[10] Strona domowa serwera J2EE firmy Weblogic, http://www.weblogic.com

[11] Strona opisująca klaster zaimplementowany w Tomcat, wersji 5.0, http://jakarta.apache.org/tomcat/tomcat-5.0-doc/cluster-howto.html