Serwer 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).

- 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].

- 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. recilmś). 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. DataSourc) – 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.

- 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.

- Hierarchia obiektów

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

– Korzeniem drzewa jest maszyna (obiekt implementujący interfejs org. apache. catalina. Engin). Reprezentuje ona niezależny serwer.

- Maszyna posiada węzły (obiekty implementujące interfejs apache. catalina.Host), które reprezentująwirtualne węzły.
- Węzeł posiada konteksty (obiekty implementujące interfejs apache. catalina. Contexi), które reprezentują aplikacje internetowe (ang. w eh 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 apache. catalina. Manage) 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.

- 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.

- Klaster w serwerze Tomcat

W wersji 5.0 została dodana obsługa klastra. Administrator może zdefiniować klasę implementującą klaster (interfejs org. apache. catalina. ciuster) 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 XXXjest nazwą parametru oraz jednocześnie nazwą atrybutu w pliku XML). Dodatkowo w specjalny sposób traktowane są obiekty implementujące interfejs klasy org.apache . catalina . Lifecycle.

- Obiekty klasy Lifecycle

Jeżeli tworzone na poziomie serwera obiekty implementują interfejs org.apache. catalina. Lifecycl , to w momencie startu systemu wywoływana jest dla nich metoda starto. Obiekty tego interfejsu zostaną powiadomione o zamknięciu systemu poprzez wywołanie dla nich metody stop(). Dla przykładu obiekt implementujący interfejs ciuster w module klastrowania jednocześnie implementuje interfejs Lifecycl , aby w metodach start i stój 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.

- 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.

- 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 (Reąuest, 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 (Reąuest, 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.

- 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 rozgłoszeni owym (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.

- Architektura klastra w wersji 5.0

Klaster jest całkowicie zdecentralizowany, każdy kolejny węzeł dołącza się rozsyłając odpowiedni komunikat w trybie rozgłoszeni owym, 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.

- Implementacja klastra w wersji 5.0

Kody źródłowe modułu można podzielić na trzy części:

1. warstwa transportuj ąca (por. p. 3.4.3.1),
2. warstwa związana z serwerem Tomcat: menedżer sesji, obiekt sesji (por. p. 3.4.32),
3. klasy pomocnicze (por. p. 3.4.3.3).

- 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 3essionMessage. 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_s e s s i on s – nowy węzeł pobiera wszystkie do tej pory stworzone sesje;
6. EVT_ALL_SESsioN_DATi – 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 (Objectlnputstream stream). Przy odtwarzaniu danych sesji z tablicy bajtów, tworzony jest strumień wejściowy Replicationstrea , 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.

- Warstwa związana z serwerem Tomcat

Moduł Hanika podłączany jest za pomocą klasy simpleTcpCluste , którą definiuje się w znaczniku <ciuster: , w pliku konfiguracyjnym serwera. Klasa implementuje standardowy interfejs org. apache. catalina. ciuste:, 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 starto 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 SimpleTcpReplicationManagei, 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

reąuestcompieted(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.

- 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.

- 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. mcastFreąuency – 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 wyj ścia,
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 iseDirty będzie ustawiona na fałsz.

- 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.

- 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.

W serwisie dyplom.com.pl prezentujemy obronione prace dyplomowe, które mogą służyć za wzór do napisania własnej pracy - gdyby potrzebowali jeszcze Państwo konsultacji to polecamy stronę pisanie prac - fachowa pomoc w pisaniu prac.