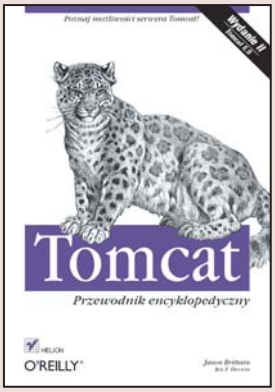
Tomcat. Przewodnik encyklopedyczny. Wydanie II



Autor: Jason Brittain, Ian Darwin
Tłumaczenie: Piotr Pilch
ISBN: 978-83-246-1594-0
Tytuł oryginału: Tomcat: The Definitive Guide, 2nd edition
Stron: 490

Poznaj możliwości serwera Tomcat!

- Jak dostroić Tomcat w celu pomiaru i poprawy wydajności?
- Jak wdrażać aplikacje WWW z serwletami i stronami JSP?
- Jak diagnozować problemy z serwerem?

Tomcat jest kontenerem serwletów Java i serwerem WWW stworzonym przez organizację Apacze Software Foundation. Może pełnić rolę serwera produkcyjnego o dużej wydajności, sprawdza się również jako darmowy kontener serwletów i stron JSP z udostępnionym kodem źródłowym. Tomcat może być zastosowany niezależnie lub w połączeniu z innymi serwerami WWW (np. httpd Apache). Doskonale radzi sobie w każdego rodzaju środowisku, zapewniając fundament wymagany do praktycznego wykorzystania w Internecie umiejętności z zakresu technologii Java.

W książce „Tomcat. Przewodnik encyklopedyczny" znajdziesz szczegółowe wyjaśnienia, jak korzystać z tego serwera. Czytając ją, poznasz wszelkie procedury instalacyjne oraz możliwości konfigurowania obszarów, ról, użytkowników i zasobów JNDI. Nauczysz się, jak uaktywniać i wyłączać funkcję automatycznego przeładowywania serwletów, a także wdrażać aplikacje WWW. Niezbędne informacje dotyczące serwera Tomcat znajdą tu nie tylko programiści, ale także administratorzy, webmasterzy i wszyscy, którzy chcą się dowiedzieć czegoś o tym kontenerze serwletów.

- Instalowanie i konfigurowanie Tomcata
- Zarządzanie obszarami, rolami i użytkownikami
- Uruchamianie i zatrzymywanie serwera
- Kontrolowanie i utrwalanie sesji
- Optymalizowanie wydajności serwera
- Integracja z serwerem WWW Apache
- Wdrażanie rozpakowanego katalogu aplikacji WWW
- Praca z plikami WAR
- Zabezpieczenia serwera Tomcat

Przewodnik dla wszystkich, którzy chcą ułatwić sobie pracę z serwerem Tomcat

Wydawnictwo Helion
ul. Kościuszki 1c
44-100 Gliwice
tel. 032 230 98 63
e-mail: helion@helion.pl

Spis treści

ROZDZIAŁ 4.

Optymalizowanie wydajności serwera Tomcat

Po zainstalowaniu i uruchomieniu serwera Tomcat Czytelnik prawdopodobnie będzie chciał zoptymalizować jego wydajność, żeby efektywniej obsługiwał żądania trafiające do komputera. W tym rozdziale przedstawimy kilka pomysłów dotyczących optymalizowania wydajności środowiska uruchomieniowego i samego serwera Tomcat.

Sztuka dostrajania serwera jest złożonym zadaniem. Składa się z pomiaru, analizy, modyfikacji i ponownie pomiaru. Oto podstawowe kroki procesu optymalizowania:

1. Zdecydowanie, co ma być zmierzone.
2. Określenie metody pomiaru.
3. Pomiar.
4. Przeanalizowanie wniosków wynikających z uzyskanych informacji.
5. Zmodyfikowanie konfiguracji przy wykorzystaniu metod, które powinny poprawić osiągi.
6. Pomiar i porównanie wyników z poprzednio uzyskanymi.
7. Ponowne zrealizowanie kroku 4.

Warto zauważyć (co zresztą widać), że nie jest dostępna klauzula „wyjścia z pętli" (być może odzwierciedlająca rzeczywistość). W praktyce trzeba będzie określić próg, poniżej którego mniej istotne zmiany będą tak mało znaczące, że będzie można zająć się innymi codziennymi zmartwieniami. Dostosowywanie i pomiar można zakończyć, gdy uzyska się przekonanie, że wystarczająco bliskie są czasy odpowiedzi, które spełnią postawione wymagania.

Aby zdecydować, co należy zoptymalizować w celu osiągnięcia lepszej wydajności, powinno się przeprowadzić niżej opisane działania.

Na komputerze testowym należy uruchomić serwer Tomcat tak samo skonfigurowany jak w przypadku środowiska produkcyjnego. Warto zastosować taki sam sprzęt, system operacyjny, bazę danych itp. Im bardziej środowisko testowe będzie przypominać produkcyjne, tym większe będą szanse zidentyfikowania wąskich gardeł, które pojawią się w konfiguracji środowiska produkcyjnego.

Na oddzielnym komputerze należy zainstalować i skonfigurować generator obciążenia i oprogramowanie mierzące czasy odpowiedzi, które posłuży do testowania obciążenia. Jeśli oprogramowanie uruchomi się na tym samym komputerze co serwer Tomcat, wyniki testów będą nie do końca precyzyjne, a czasami nieprawdziwe. W idealnej sytuacji Tomcat powinien działać na jednym komputerze, a oprogramowanie testujące na innym. Jeżeli nie dysponuje się wystarczającą liczbą komputerów, nie pozostaje nic innego, jak całe oprogramowanie załadować na testowym komputerze. Testy przeprowadzone w ten sposób będą lepsze od zupełnego zrezygnowania z nich. Jednak uruchomienie na tym samym komputerze klienta sprawdzającego obciążenie i serwera Tomcat spowoduje, że uzyska się krótsze czasy odpowiedzi, które przy kolejnych powtórzeniach tego samego testu okażą się mniej zgodne.

Należy wyizolować komunikację między komputerem testującym obciążenie i komputerem, na którym uruchomiono serwer Tomcat. Jeśli przeprowadza się intensywne testy, nie będzie pożądane zniekształcanie ich danych przez ruch sieciowy niestanowiący części testów. Ponadto nie będzie mile widziane obciążanie komputerów niezaangażowanych w testy na skutek dużego ruchu sieciowego generowanego przez testy. Między komputerem testującym i serwerem produkcyjnym należy umieścić przełącznik lub zastosować koncentrator, do którego podłączono tylko te dwa komputery.

Należy przeprowadzić kilka testów obciążenia, symulujących różnego typu sytuacje charakteryzujące się dużym ruchem sieciowym, które mogą wystąpić w przypadku serwera produkcyjnego. Aby lepiej przygotować się na przyszłą rozbudowę środowiska, dodatkowo powinno się prawdopodobnie wykonać kilka testów generujących ruch sieciowy *większy* od oczekiwanego w przypadku serwera produkcyjnego.

Należy zidentyfikować wszelkie nietypowo długie czasy odpowiedzi i spróbować stwierdzić, jakie składniki sprzętowe i (lub) programowe są tego przyczyną. Zazwyczaj odpowiada za to oprogramowanie. Jest to dobra wiadomość, gdyż w pewnym stopniu problem z długim czasem odpowiedzi można zmniejszyć przez przekonfigurowanie lub przebudowanie aplikacji. Jednak w ekstremalnych przypadkach może być konieczne użycie dodatkowego sprzętu lub nowszych, szybszych i kosztowniejszych urządzeń. Obserwować należy średnie obciążenie komputera serwera, a także w plikach dzienników Tomcata szukać komunikatów o błędzie.

W niniejszym rozdziale zaprezentujemy niektóre z typowych ustawień serwera Tomcata kwalifikujących się do dostrojenia, w tym związane z wydajnością serwera WWW, pulą wątków żądań Tomcata, wydajnością wirtualnej maszyny Java, konfiguracją sprawdzania adresów usługi DNS i wstępną kompilacją stron JSP. Na końcu rozdziału wspomnimy o planowaniu obciążenia.

Pomiar wydajności serwera WWW

Pomiar wydajności serwera WWW jest groźnie wyglądającym zadaniem, któremu w tym miejscu powinniśmy poświęcić trochę uwagi i podać odnośniki do bardziej obszernych prac poświęconych tej tematyce. Z wydajnością serwera WWW jest związanych zbyt wiele zmiennych, żeby w pełni omówić to zagadnienie. Większość strategii pomiaru wykorzystuje program-klienta, który pełni rolę przeglądarki, lecz w rzeczywistości mniej więcej w tym samym czasie wysyła ogromną liczbę żądań i mierzy czasy odpowiedzi[1].

[1] Istnieje też rozwiązanie serwerowe polegające na uruchamianiu Tomcata pod kontrolą narzędzia Java Profiler w celu zoptymalizowania kodu serwera. Jednak będzie to bardziej interesujące dla programistów niż administratorów.

Trzeba zdecydować, jak zostanie przeprowadzony test i co dokładnie będzie sprawdzane — czy na przykład powinno się na tym samym komputerze uruchamiać klienta testującego obciążenie i pakiety oprogramowania serwerowego. Szczególnie odradzamy robienie czegoś takiego. Załadowanie klienta i serwera na tym samym komputerze spowoduje zmienności i niestabilność uzyskiwanych wyników. Czy w chwili wykonywania testów na komputerze serwera coś innego jest aktywne? Czy klient i serwer powinny być połączone ze sobą za pośrednictwem łącza Gigabit Ethernet, 100baseT czy 10baseT? Z doświadczenia wiemy, że jeśli klient testujący obciążenie jest podłączony do komputera serwera za pomocą łącza wolniejszego niż Gigabit Ethernet, samo połączenie sieciowe może spowolnić testy, a tym samym zmienić ich rezultat.

Czy klient powinien wielokrotnie żądać tej samej strony, jednocześnie generować kilka różnego typu żądań, czy losowo wybierać stronę z dużej listy stron? Może to mieć wpływ na wydajność serwera dotyczącą buforowania i wielowątkowości. To, jak się w tym przypadku postąpi, będzie zależeć od rodzaju symulowanego klienta obciążenia. Jeżeli symulowane są działania użytkowników, prawdopodobnie będą oni żądali różnych stron, a nie jednej za każdym razem. Jeśli symuluje się programowego klienta HTTP, może on wielokrotnie żądać tej samej strony. W związku z tym klient testujący powinien raczej robić to samo. Najpierw należy określić typ ruchu sieciowego generowanego przez klienty, a następnie tak skonfigurować klienty testujące obciążenie, żeby zachowywały się jak klienty rzeczywiste.

Czy klient testujący powinien wysyłać żądania regularnie czy seriami? Gdy podczas pomiarów chce się stwierdzić, jak szybko serwer jest w stanie w pełni obsłużyć żądania, powinno się spowodować, żeby klient testujący wysyłał żądania jedno po drugim bez żadnych przerw między kolejnymi. Czy aktywny serwer korzysta z ostatecznej konfiguracji, czy w dalszym ciągu jest realizowany proces debugowania, który może generować dodatkowe obciążenie? W czasie pomiarów powinno się wyłączyć wszystkie funkcje procesu debugowania. Można również zrezygnować z części operacji rejestrowania. Czy klient HTTP powinien żądać obrazów, czy po prostu strony HTML z osadzonymi obrazami? Zależy to od tego, jak dokładnie zamierza się symulować ruch generowany przez użytkowników korzystających z witryn internetowych. Mamy nadzieję, że Czytelnik doszedł do następującego wniosku: istnieje wiele różnego typu testów wydajności, które można przeprowadzić, i każdy z nich da odmienne (i raczej interesujące) wyniki.

Narzędzia testujące obciążenie

Zadaniem postawionym przed większością narzędzi mierzących obciążenie związane z witrynami internetowymi jest zażądanie jednego lub większej liczby zasobów serwera WWW określoną (dużą) liczbę razy i dokładne poinformowanie użytkownika, ile czasu proces ten potrwał z punktu widzenia klienta (lub ile razy w ciągu sekundy strona mogła zostać pobrana). W internecie jest dostępnych wiele narzędzi mierzących obciążenie witryn WWW (pod adresem *http://www.softwareqatest.com/qatweb1.html#LOAD* znajduje się lista z niektórymi z tych narzędzi). Wśród godnych uwagi narzędzi pomiarowych należy wymienić takie, jak Apache Benchmark (program ab dołączony do dystrybucji serwera *httpd* Apache dostępnego pod adresem *http://httpd.apache.org*), siege (*http://www.joedog.org/JoeDog/Siege*) i JMeter wchodzący w skład oprogramowania Apache Jakarta (*http://jakarta.apache.org/jmeter*).

Z tych trzech narzędzi testujących obciążenie najwięcej funkcji oferuje program JMeter. Narzędzie to napisano w czystym wieloplatformowym języku Java. Cechuje się ładnym graficznym

interfejsem użytkownika, który jest wykorzystywany zarówno do konfigurowania, jak i generowania wykresów obciążenia. Program JMeter jest bogaty w możliwości i wyróżnia się elastycznością w zakresie testowania witryn i generowania raportów. Można go uruchomić w trybie tekstowym. Do narzędzia dołączono szczegółową dokumentację sieciową, objaśniającą, jak je skonfigurować i obsługiwać. Z doświadczenia wiemy, że program JMeter zapewnia dla wyników testów większość opcji raportowania, cechuje się największą przenośnością na różne systemy operacyjne i obsługuje większość funkcji. Jednak z jakiegoś powodu nie mógł generować w ciągu sekundy tak wielu żądań HTTP jak narzędzia ab i siege. Jeśli nie próbuje się stwierdzić, ile serwer Tomcat może obsłużyć żądań na sekundę, program JMeter dobrze się sprawdzi, gdyż prawdopodobnie zastosuje wszystkie funkcje wymagane przez użytkownika. Jeżeli jednak podejmie się próbę określenia maksymalnej liczby żądań z powodzeniem zrealizowanych w ciągu sekundy przez serwer, powinno się sięgnąć po narzędzie ab lub siege.

Jeśli szuka się narzędzia pomiarowego trybu wiersza poleceń, program ab sprawdzi się w tej roli znakomicie. Ponieważ jest to wyłącznie narzędzie pomiarowe, raczej nie użyje się go do przeprowadzenia testów regresji. Choć program ab nie oferuje graficznego interfejsu użytkownika i jednocześnie nie można mu przekazać listy liczącej więcej niż jeden adres URL, wyjątkowo dobrze radzi sobie z wykonaniem pomiaru dla jednego adresu, a następnie zaprezentowaniem precyzyjnych i szczegółowych wyników. W większości systemów operacyjnych innych niż Windows narzędzie ab jest domyślnie instalowane z serwerem *httpd* Apache. Można też skorzystać z oficjalnego pakietu serwera *httpd* Apache zawierającego narzędzie ab, dzięki czemu będzie ono najprostsze do zainstalowania spośród wszystkich narzędzi testujących obciążenie witryn internetowych.

`siege` jest następnym dobrym narzędziem trybu wiersza poleceń (bez graficznego interfejsu), które testuje obciążenie witryn internetowych. Choć w przypadku większości systemów operacyjnych program siege nie jest domyślnie instalowany, instrukcje dotyczące procesu jego budowania i instalowania są zrozumiałe i wyjątkowo proste. Kod narzędzia siege napisany w języku C cechuje się dużą przenośnością. Program obsługuje wiele różnych metod uwierzytelniania, może przeprowadzić test porównawczy i regresji, a także oferuje tryb internetowy, w którego przypadku próbuje bardziej dokładnie symulować obciążenie aplikacji WWW generowane przez wielu rzeczywistych użytkowników internetowych. Wydaje się, że inne narzędzia, o mniejszych możliwościach, kiepsko obsługują uwierzytelnianie aplikacji WWW. Programy te umożliwiają wysyłanie plików *cookie*, lecz część z nich może nie pozwalać na odbieranie tych plików. Choć serwer Tomcat obsługuje kilka różnych metod uwierzytelniania (podstawowe, szyfrowane, formularza i z zastosowaniem certyfikatu klienta), część z tych prostszych narzędzi jest zgodna jedynie z podstawowym uwierzytelnianiem HTTP. Uwierzytelnianie formularza może być przetestowane przez dowolne narzędzie mogące wysyłać formularz (zależy to od tego, czy program obsługuje przekazywanie żądań `POST` HTTP dotyczących przesłania formularza logowania; żądania `POST` mogą być wysyłane przez narzędzia JMeter, ab i siege). Coś takiego jest możliwe tylko w przypadku niektórych z bardziej ograniczonych narzędzi. Możliwość dokładniejszego symulowania uwierzytelniania użytkownika w środowisku produkcyjnym jest istotnym elementem testów wydajności, gdyż sam proces uwierzytelniania często generuje duże obciążenie i zmienia charakterystyki wydajnościowe witryny internetowej. Zależnie od metody uwierzytelniania wykorzystywanej w środowisku produkcyjnym, może być konieczne poszukanie innych narzędzi, które metodę tę obsługują.

Gdy książkę przygotowano do druku, pojawił się nowy pakiet oprogramowania porównawczego o nazwie Faban (*http://faban.sunsource.net*). Narzędzie Faban zostało stworzone przez firmę

Sun Microsystems w czystym języku Java 1.5+. Jest to oprogramowanie *open source* objęte licencją CDDL. Narzędzie Faban wydaje się skoncentrowane wyłącznie na dokładnym porównywaniu różnego typu serwerów, w tym serwerów WWW. Ponieważ narzędzie to zostało dopracowane w celu zapewnienia wysokiej wydajności i krótkich czasów odpowiedzi, wszelkie pomiary będą jak najbardziej zbliżone do rzeczywistych osiągów serwera. Dane pomiarowe są na przykład zbierane, gdy nie jest wykonywany żaden inny kod narzędzia Faban, a analiza danych ma miejsce tylko po zakończeniu pomiarów. Aby uzyskać jak najlepszą dokładność, właśnie w taki sposób wszystkie testy pomiarowe powinny być realizowane. Oprogramowanie Faban dysponuje również bardzo ładną konsolą konfiguracyjną i administracyjną w postaci aplikacji WWW. W celu obsługi tej konsoli narzędzie Faban jest zintegrowane z serwerem Tomcat! Tak, Tomcat stanowi część programu Faban. Każdy projektant używający języka Java, zainteresowany zarówno Tomcatem, jak i pomiarami wydajności, może zapoznać się z dokumentacją i kodem źródłowym Fabana, a także opcjonalnie uczestniczyć w procesie tworzenia oprogramowania Faban. Jeśli Czytelnik programuje w języku Java i szuka długoterminowego rozwiązania porównawczego, cechującego się jak największą liczbą możliwości, prawdopodobnie Faban jest tym, co powinno się zastosować. Choć nie dysponowaliśmy ilością czasu pozwalającą nam na opisanie w książce narzędzia Faban w szerszym zakresie, na szczęście jego witryna udostępnia znakomitą dokumentację.

ab — narzędzie serwera Apache mierzące wydajność

Narzędzie ab pobiera pojedynczy adres URL i żąda go każdorazowo, korzystając z liczby wątków określonych przez użytkownika. ab oferuje różne argumenty wiersza poleceń kontrolujące liczbę pobrań adresu URL i pozwala ustalić maksymalną liczbę jednoczesnych wątków. Wśród kilku przydatnych funkcji programu ab można wyróżnić opcjonalną możliwość okresowego drukowania raportów dotyczących postępu operacji i kompletnych raportów.

Przykład 4.1 prezentuje zastosowanie narzędzia ab. Zostało ono poinstruowane do pobrania adresu URL 100 000 razy z jednoczesnym wykorzystaniem 149 wątków. Ustawiliśmy takie wartości z rozwagą. Im mniejsza liczba żądań HTTP tworzonych przez klienta testującego w czasie testu porównawczego, tym większe prawdopodobieństwo tego, że klient poda mniej dokładne wyniki. Wynika to stąd, że podczas testu porównawczego wstrzymywanie procesu przez składnik wirtualnej maszyny Java odpowiedzialny za zbieranie zwolnionych obszarów pamięci (ang. *garbage collector*) powoduje zwiększenie procentowego udziału tego składnika w całkowitym czasie trwania testu. Im wyższa całkowita liczba wygenerowanych żądań HTTP, tym mniej znaczące stają się przerwy wywołane przez składnik zbierający zwolnione obszary pamięci i tym bardziej prawdopodobne jest to, że wyniki pomiarów pokażą, jak ogólnie działa serwer Tomcat. Testy wydajności powinno się przeprowadzać przez utworzenie co najmniej 100 000 żądań HTTP. Poza tym można tak skonfigurować klienta testującego, żeby wywoływał żądaną liczbę wątków. Jednak nie uzyska się pomocnych wyników, jeśli ustawiona liczba wątków przekroczy wartość argumentu `maxThreads` określonego w obrębie elementu `Connector` znajdującego się w pliku *conf/server.xml* serwera Tomcat. Domyślnie wartością tego argumentu jest `150`. Jeżeli dla klienta testującego ustawi się wyższą wartość i wygeneruje się więcej żądań w liczbie wątków przekraczającej liczbę wątków Tomcata, które odbierają i przetwarzają te żądania, wydajność spadnie, gdyż część wątków z żądaniami klienta zawsze będzie musiała czekać. Najlepiej po prostu ustawić liczbę wątków klienta mniejszą od wartości atrybutu `maxThreads` elementu `Connector` (na przykład 149).

Przykład 4.1. Pomiar wydajności za pomocą narzędzia ab

```
$ ab -k -n 100000 -c 149 http://host_tomcata:8080
This is ApacheBench, Version 2.0.40-dev <$Revision$> apache-2.0
Copyright 1996 Adam Twiss, Zeus Technology Ltd, http://www.zeustech.net/
Copyright 1997-2005 The Apache Software Foundation, http://www.apache.org/

Benchmarking tomcathost (be patient)
Completed 10000 requests
Completed 20000 requests
Completed 30000 requests
Completed 40000 requests
Completed 50000 requests
Completed 60000 requests
Completed 70000 requests
Completed 80000 requests
Completed 90000 requests
Finished 100000 requests

Server Software:        Apache-Coyote/1.1
Server Hostname:        tomcathost
Server Port:            8080

Document Path:          /
Document Length:        8132 bytes

Concurrency Level:      149
Time taken for tests:   19.335590 seconds
Complete requests:      100000
Failed requests:        0
Write errors:           0
Keep-Alive requests:    79058
Total transferred:      830777305 bytes
HTML transferred:       813574072 bytes
Requests per second:    5171.81 [#/sec] (mean)
Time per request:       28.810 [ms] (mean)
Time per request:       0.193 [ms] (mean, across all concurrent requests)
Transfer rate:          41959.15 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        0    1   4.0      0      49
Processing:     2   26   9.1     29      62
Waiting:        0   12   6.0     13      40
Total:          2   28  11.4     29      65

Percentage of the requests served within a certain time (ms)
  50%     29
  66%     30
  75%     31
  80%     45
  90%     47
  95%     48
  98%     48
  99%     49
 100%     65 (longest request)
```

Jeśli w wierszu polecenia `ab` nie umieści się argumentu `-k`, narzędzie nie będzie korzystać z ciągle aktywnych połączeń z serwerem Tomcat. Jest to mniej efektywne rozwiązanie, ponieważ w celu utworzenia każdego żądania HTTP narzędzie ab musi połączyć się z Tomcatem przy użyciu nowego gniazda TCP. W rezultacie w ciągu sekundy może być obsłużona mniejsza liczba żądań, a ponadto będzie mniejsza przepustowość między serwerem Tomcat i klientem (programem ab) (przykład 4.2).

Przykład 4.2. Pomiar wydajności za pomocą narzędzia ab z wyłączoną funkcją ciągle aktywnych połączeń

```
$ ab -n 100000 -c 149 http://host_tomcata:8080/
This is ApacheBench, Version 2.0.40-dev <$Revision$> apache-2.0
Copyright 1996 Adam Twiss, Zeus Technology Ltd, http://www.zeustech.net/
Copyright 1997-2005 The Apache Software Foundation, http://www.apache.org/

Benchmarking tomcathost (be patient)
Completed 10000 requests
Completed 20000 requests
Completed 30000 requests
Completed 40000 requests
Completed 50000 requests
Completed 60000 requests
Completed 70000 requests
Completed 80000 requests
Completed 90000 requests
Finished 100000 requests

Server Software:        Apache-Coyote/1.1
Server Hostname:        tomcathost
Server Port:            8080

Document Path:          /
Document Length:        8132 bytes

Concurrency Level:      149
Time taken for tests:   28.201570 seconds
Complete requests:      100000
Failed requests:        0
Write errors:           0
Total transferred:      831062400 bytes
HTML transferred:       814240896 bytes
Requests per second:    3545.90 [#/sec] (mean)
Time per request:       42.020 [ms] (mean)
Time per request:       0.282 [ms] (mean, across all concurrent requests)
Transfer rate:          28777.97 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        0   18  11.3     19      70
Processing:     3   22  11.3     22      73
Waiting:        0   13   8.4     14      59
Total:         40   41   2.4     41      73

Percentage of the requests served within a certain time (ms)
  50%     41
  66%     41
  75%     42
  80%     42
  90%     43
  95%     44
  98%     46
  99%     55
 100%     73 (longest request)
```

siege

W celu zastosowania narzędzia siege do przeprowadzenia testu porównawczego takiego samego jak wyżej opisany należy użyć wiersza polecenia podobnego jak w przypadku programu ab, z tym że trzeba podać liczbę generowanych żądań, które mają przypadać *na jeden wątek*. Jeśli w ramach testu próbuje się utworzyć 100 000 żądań HTTP z wykorzystaniem jednocześnie

149 klientów, trzeba poinformować narzędzie siege, że każdy z tych klientów musi utworzyć 671 żądań (ponieważ 671 żądań pomnożonych przez 149 klientów w przybliżeniu daje 100 000 żądań). Wstawienie w wierszu polecenia `siege` argumentu `-b` spowoduje, że narzędzie przeprowadzi test porównawczy. W efekcie wątki klientów nie będą wstrzymywały pracy między kolejnymi żądaniami, jak to ma miejsce w przypadku narzędzia ab. Domyślnie program siege między kolejnymi żądaniami odczekuje określony czas. Jednak w trybie testu porównawczego tak nie jest. Przykład 4.3 prezentuje wiersz polecenia `siege` i wyniki testu porównawczego.

Przykład 4.3. Przeprowadzenie testu porównawczego za pomocą narzędzia siege, dla którego wyłączono funkcję ciągle aktywnych połączeń

```
$ siege -b -r 671 -c 149 host_tomcata:8080
** siege 2.65
** Preparing 149 concurrent users for battle.
The server is now under siege..      done.
Transactions:                  99979 hits
Availability:                 100.00 %
Elapsed time:                  46.61 secs
Data transferred:             775.37 MB
Response time:                  0.05 secs
Transaction rate:            2145.01 trans/sec
Throughput:                    16.64 MB/sec
Concurrency:                  100.62
Successful transactions:       99979
Failed transactions:               0
Longest transaction:           23.02
Shortest transaction:           0.00
```

Z wynikami narzędzia siege związanych jest kilka interesujących rzeczy, o których należy wspomnieć. Oto one:

- Liczba transakcji zrealizowanych w ciągu sekundy przez narzędzie siege jest znacznie mniejsza niż w przypadku programu ab (jest tak, gdy dla obu klientów testujących wyłączono funkcję ciągle aktywnych połączeń[2], a wszystkie pozostałe ustawienia są takie same). Jedynym wytłumaczeniem takiego stanu rzeczy jest to, że narzędzie siege nie jest tak efektywne, jak program ab. Na tej podstawie można wywnioskować, że wyniki testu porównawczego narzędzia siege nie są tak dokładne, jak wyniki zwrócone przez program ab.
- Przepustowość, o której informuje narzędzie siege, jest znacznie mniejsza od uzyskiwanej przez program ab. Prawdopodobnie jest to spowodowane tym, że narzędzie siege nie może przetworzyć w ciągu sekundy tak wiele żądań, jak program ab.
- Podawana przez narzędzie siege całkowita ilość przesłanych danych w przybliżeniu jest równa osiągom w tym przypadku programu ab.
- Program ab ukończył test porównawczy w czasie wynoszącym trochę ponad połowę czasu, jaki na to samo potrzebowało narzędzie siege. Jednak nie wiemy, ile z tego czasu narzędzie siege poświęciło na oczekiwanie między kolejnymi żądaniami każdego wątku. Może po prostu być tak, że pętla narzędzia siege obsługująca żądania nie została w takim stopniu zoptymalizowana, żeby od razu przetwarzać następne żądanie.

[2] Narzędzie siege nie może przeprowadzać testu przy włączonej funkcji ciągle aktywnych połączeń. W rzeczywistości funkcji tej program siege jest pozbawiony (przynajmniej w czasie, gdy pisano książkę). Oznacza to, że korzystając z tego narzędzia, nie można wykonywać najbardziej intensywnych wydajnościowych testów porównawczych. Narzędzie umożliwia jednak przeprowadzenie innego typu testów nieobsługiwanych przez program ab. Wśród nich należy wymienić test regresji i test trybu "", w którego przypadku narzędzie siege może generować losowe żądania klientów, żeby dokładniej symulować rzeczywiste obciążenie witryn internetowych.

W celu uzyskania jak najlepszych wyników testów porównawczych najlepiej używać narzędzia ab zamiast siege. Jednak przy innego rodzaju testach, w których przypadku trzeba dokładnie symulować ruch sieciowy generowany przez użytkowników przeglądających strony internetowe, program ab nie jest odpowiedni, gdyż nie oferuje żadnej funkcji pozwalającej określić ilość czasu oczekiwania między kolejnymi żądaniami. Narzędzie siege zapewnia funkcję powodującą odczekiwanie programu losową ilość czasu między kolejnymi żądaniami. Oprócz tego narzędzie siege może pobrać losowe adresy URL z predefiniowanej listy. Z tego powodu narzędzie siege może być użyte do symulowania obciążenia generowanego przez użytkownika. Czegoś takiego program ab nie umożliwia. Więcej informacji na temat funkcji narzędzia siege można znaleźć w jego dokumentacji (należy wykonać polecenie `man siege`).

Apache Jakarta JMeter

Narzędzie JMeter można uruchomić w trybie graficznym lub tekstowym. Choć plany testów można uaktywnić w dowolnym trybie, trzeba je utworzyć w trybie graficznym. Plany testów są przechowywane w dokumentach konfiguracyjnych XML. Jeśli w konfiguracji planu testu trzeba zmienić tylko jedną wartość numeryczną lub łańcuchową, prawdopodobnie będzie można to zrobić za pomocą edytora tekstu. Jednak z myślą o kontroli poprawności dobrym pomysłem jest edytowanie planów testów przy użyciu graficznego interfejsu narzędzia JMeter.

Zanim podejmie się próbę przeprowadzenia za pomocą programu JMeter testu porównawczego dla serwera Tomcat, trzeba upewnić się, że wirtualną maszynę Java tego narzędzia załadowano z wystarczającą ilością pamięci sterty. Jest to niezbędne do uniknięcia tego, że wirtualna maszyna Java w trakcie wykonywania testów porównawczych przez narzędzie JMeter będzie spowalniać jego pracę przez realizowanie procesu zbierania zwolnionych obszarów pamięci. Jest to szczególnie ważne, gdy testy porównawcze przeprowadza się w trybie graficznym. W skrypcie startowym *bin/jmeter* znajduje się ustawienie konfiguracyjne określające rozmiar pamięci sterty. Wygląda ono następująco:

```
# Jest to podstawowy rozmiar sterty. Można go zwiększyć lub zmniejszyć odpowiednio
# do ilości dostępnej pamięci systemowej:
HEAP="-Xms256m -Xmx256m"
```

Narzędzie JMeter użyje całej przydzielonej mu pamięci sterty. Im więcej pamięci, tym rzadziej konieczne będzie zbieranie zwolnionych obszarów pamięci. Jeżeli komputer z załadowanym programem JMeter dysponuje wystarczającą ilością pamięci, zamiast obu wartości `256` powinno się ustawić większe (na przykład `512`). Istotne jest, żeby zrobić to w pierwszej kolejności, ponieważ domyślna wartość tego ustawienia może zniekształcić wyniki testu porównawczego.

W celu utworzenia planu testu porównawczego należy najpierw program JMeter załadować w trybie graficznym.

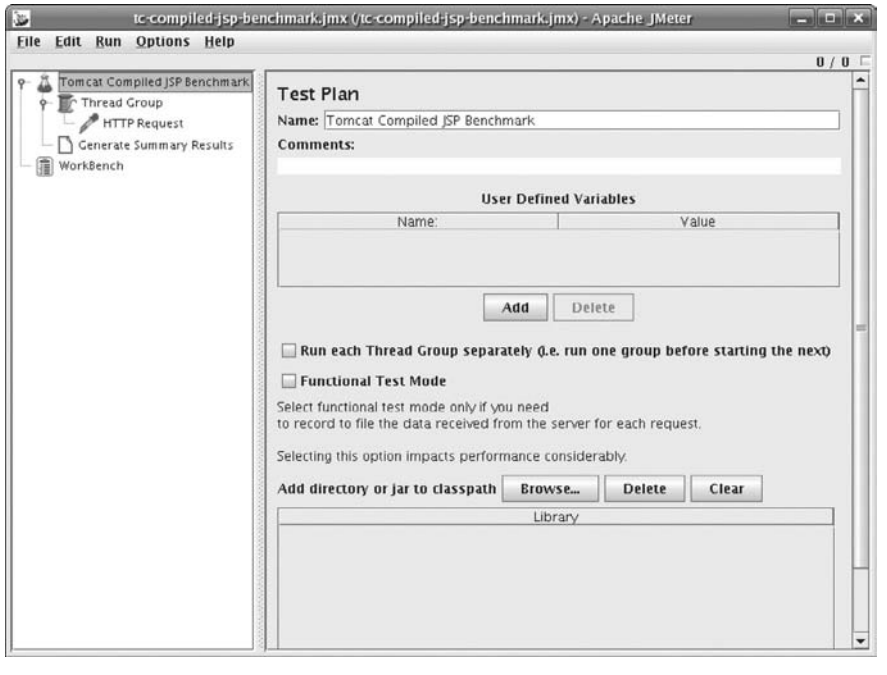
```
$ bin/jmeter
```

Okno narzędzia JMeter po lewej stronie zawiera widok drzewa, a po prawej stronie panel ze szczegółami zaznaczonego elementu. Po wybraniu czegoś w obrębie widoku drzewa w panelu pojawią się dokładne informacje dotyczące konkretnej pozycji. Aby przeprowadzić dowolny test, wewnątrz drzewa trzeba połączyć i skonfigurować odpowiednie obiekty, a następnie narzędzie JMeter może wykonać test i przekazać wyniki.

W celu przygotowania testu porównawczego, takiego jak przeprowadzony za pomocą narzędzi ab i siege, należy wykonać następujące kroki:

1. W obrębie widoku drzewa prawym przyciskiem myszy kliknąć węzeł *Test Plan* i wybrać polecenie *Add/Thread Group*.
2. W panelu szczegółów obiektu *Thread Group*, w polach *Number of Threads (users)*, *Ramp-Up Period (in seconds)* i *Loop Count*, ustawić odpowiednio wartości `149`, `0` i `671`.
3. Prawym przyciskiem myszy kliknąć węzeł drzewa *Thread Group* i z menu wybrać polecenie *Add/Sampler/HTTP Request*.
4. W panelu szczegółów obiektu *HTTP Request* zmienić ustawienia *Web Server* tak, żeby identyfikowały serwer Tomcat i numer jego portu. Dodatkowo w polu *Path* należy określić adres URI zasobu instalacji serwera Tomcat (na przykład katalog `/`), dla którego ma być przeprowadzony test porównawczy.
5. Prawym przyciskiem myszy ponownie kliknąć węzeł drzewa *Thread Group* i z menu wybrać polecenie *Add/Post Processors/Generate Summary Results*.
6. Ze znajdującego się na samej górze menu *File* wybrać polecenie *Save Test Plan* i wprowadzić nazwę, pod jaką ma być zapisany plan testu. Rozszerzeniem pliku planu testu programu JMeter jest *.jmx*. Rozszerzenie to nieszczęśliwie przypomina niezwiązane z nim rozszerzeniem JMX (*Java Management eXtension*).

Rysunek 4.1 przedstawia graficzny interfejs użytkownika narzędzia JMeter z zestawionym planem testu gotowym do uruchomienia. Widok drzewa znajduje się z lewej strony, a panel szczegółów z prawej.



Rysunek 4.1. Graficzny interfejs użytkownika narzędzia Apache JMeter z kompletnym planem testu

Po utworzeniu i zapisaniu planu testu można rozpocząć test porównawczy. Z menu rozwijanego *File* należy wybrać polecenie *Exit*, żeby zamknąć graficzny interfejs narzędzia JMeter. Dalej należy je załadować w trybie tekstowym wiersza poleceń, żeby przeprowadzić test porównawczy.

```
$ bin/jmeter -n -t test-strony-domowej-tc.jmx
Created the tree successfully
Starting the test
Generate Summary Results = 99979 in  71.0s = 1408.8/s Avg:    38 Min:     0 Max:
25445 Err:     0 (0.00%)
Tidying up ...
... end of run
```

Warto zauważyć, że liczba żądań na sekundę podana przez narzędzie JMeter (średnia liczba żądań w ciągu sekundy wynosi `1408,8`) jest znacznie mniejsza od raportowanej zarówno przez program ab, jak i siege (w przypadku tej samej konfiguracji sprzętowej, identycznej wersji Tomcata i testu porównawczego). Pokazuje to, że klient HTTP narzędzia JMeter jest wolniejszy od klientów programów ab i siege. Za pomocą narzędzia JMeter można stwierdzić, czy zmiana dokonana w aplikacji WWW, instalacji serwera Tomcat lub wirtualnej maszynie Java skróciła czy wydłużyła czas odpowiedzi witryn internetowych. Jednak nie można zastosować narzędzia JMeter do określenia maksymalnej liczby żądań na sekundę, które serwer może z powodzeniem obsłużyć, ponieważ klient HTTP programu okazuje się wolniejszy od kodu serwera Tomcat.

W przypadku narzędzia JMeter można też wyniki testu zaprezentować na wykresie. W tym celu ponownie należy załadować program w trybie graficznym, po czym wykonać następujące kroki:

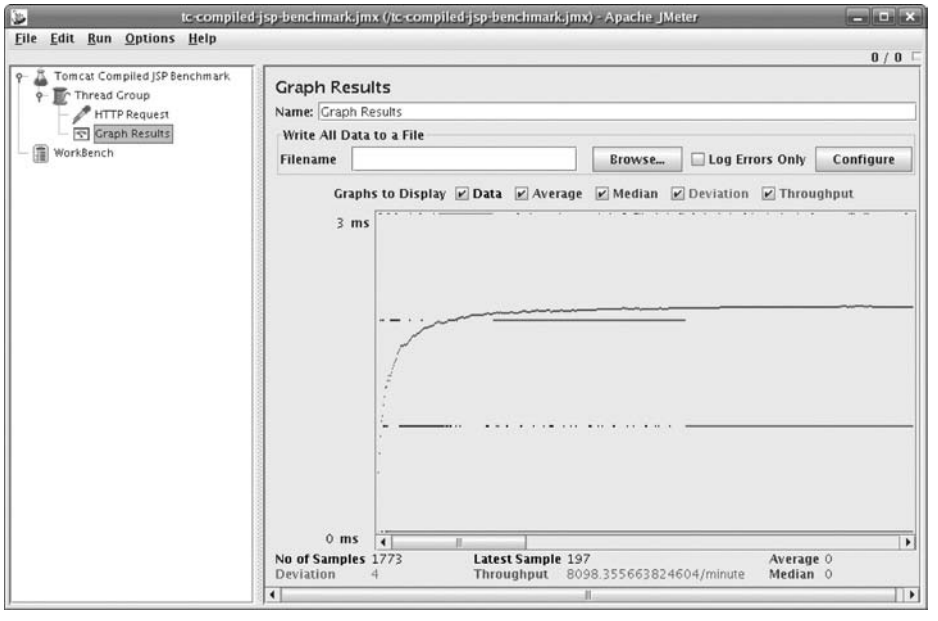
1. Otworzyć wcześniej utworzony plan testu.
2. W obrębie widoku drzewa zaznaczyć węzeł *Generate Summary Results* i usunąć go (prosta umożliwiająca to metoda polega na jednokrotnym wciśnięciu klawisza *Delete*).
3. Zaznaczyć węzeł drzewa *Thread Group*, a następnie kliknąć go prawym przyciskiem myszy i z menu wybrać polecenie *Add/Listener/Graph Results*.
4. Zapisać plan testu pod nową nazwą (tym razem w celu obejrzenia wyników testu w postaci graficznej).
5. Zaznaczyć węzeł drzewa *Graph Results*.

Można ponownie przeprowadzić test i w czasie rzeczywistym obserwować jak narzędzie JMeter wyświetla wyniki na wykresie.

Ponownie trzeba sprawdzić, czy wirtualnej maszynie Java narzędzia JMeter przydzielono wystarczającą ilość pamięci sterty, żeby w czasie trwania testu nie przeprowadzała często procesu zbierania zwolnionych obszarów pamięci. Ponadto trzeba pamiętać o tym, że wirtualna maszyna Java w czasie wykonywania testu musi poświęcić czas na generowanie wykresu. Na skutek tego zmniejszy się dokładność wyników testu. To, o ile spadnie dokładność, zależy od szybkości komputera z załadowanym narzędziem JMeter (im szybszy, tym lepiej). Jeśli jednak wykres jest rysowany tylko w celu obserwacji w czasie rzeczywistym wyników przeprowadzanego testu, jest to znakomity sposób umożliwiający to.

Gdy jest się gotowym do uruchomienia testu, z umieszczonego na samej górze okna menu rozwijanego *Run* należy wybrać polecenie *Start* lub wcisnąć kombinację klawiszy *Ctrl+R*.

Choć test porównawczy zostanie ponownie rozpoczęty, wyniki pokazywane na wykresie pojawią się, gdy dane wyjściowe zostaną zebrane przez narzędzie JMeter. Rysunek 4.2 pokazuje wykres z wynikami testu wyświetlony w graficznym interfejsie narzędzia JMeter.



Rysunek 4.2. Wyświetlanie wyników testu na wykresie generowanym przez narzędzie Apache JMeter

Można zezwolić na dokończenie testu lub zatrzymać go przez wciśnięcie kombinacji klawiszy *Ctrl+.* (przy wciśniętym klawiszu *Ctrl* należy wcisnąć klawisz kropki). Jeśli test zostanie przerwany w początkowej fazie, prawdopodobnie narzędziu JMeter kilka sekund zajmie jego zakończenie i usunięcie wszystkich wątków znajdujących się w obiekcie *Thread Group*. Aby wyczyścić wykres przed ponownym uruchomieniem testu, należy wcisnąć kombinację klawiszy *Ctrl+E*. Można też usunąć wykres w trakcie trwania testu. W tym przypadku test będzie kontynuowany, a wykres rysowany, począwszy od bieżącej próbki.

Przez zastosowanie narzędzia JMeter do wyświetlania wyników na wykresie uzyskuje się wgląd do uruchomionego testu. Dzięki temu można go obserwować i usunąć wszelkie zaistniałe problemy, a także dostosować test do własnych wymagań przed wywołaniem go z poziomu wiersza poleceń. Po uznaniu, że test został poprawnie przygotowany, należy zapisać jego plan, który nie zawiera węzła drzewa *Graph Results*, lecz uwzględnia węzeł *Generate Summary Results*, umożliwiający wywołanie testu z poziomu wiersza poleceń. Plan testu należy ponownie zapisać pod nową nazwą, opisującą rodzaj testu uruchamianego z poziomu wiersza poleceń. Jako ostateczne wyniki testu należy zastosować wyniki uzyskane z poziomu wiersza poleceń. Narzędzie testujące ab zapewnia dokładniejsze wyniki testu porównawczego, lecz nie oferuje tak wielu funkcji jak program JMeter.

Narzędzie JMeter dysponuje też znacznie większą liczbą funkcji, które mogą pomóc testować aplikacje WWW na kilka sposobów. Więcej informacji o tym znakomitym narzędziu testującym można znaleźć w dokumentacji sieciowej (*http://jakarta.apache.org/jmeter*).

Porównanie wydajności serwerów WWW

Wcześniej w rozdziale omówiono kilka klientów HTTP przeprowadzających testy porównawcze. W tym punkcie przedstawimy praktyczny przykład dotyczący serwera Tomcat, który od początku do końca demonstruje procedurę przeprowadzania testu porównawczego, a także przekazuje informacje mogące pomóc skonfigurować Tomcata tak, żeby oferował aplikacji WWW lepszą wydajność.

Testy porównawcze wykonaliśmy dla wszystkich implementacji serwera WWW Tomcata, a także niezależnego serwera *httpd* Apache i jego modułów, które łączą się z Tomcatem w celu ustalenia, jak efektywna jest każda konfiguracja w przypadku udostępniania statycznej zawartości. Czy serwer *httpd* Apache jest szybszy niż serwer Tomcat? Jaka jest najszybsza implementacja złącza serwera WWW Tomcata? Jaka jest najszybsza implementacja złącza serwera wykorzystującego protokół AJP (*Apache JServe Protocol*)? O ile każdy z serwerów jest szybszy lub wolniejszy? Odpowiemy na te pytania, testując różne konfiguracje (przynajmniej dla jednej kombinacji sprzętu, systemu operacyjnego i środowiska Java).

Ponieważ wyniki testu porównawczego w dużej mierze zależą od sprzętu, na którym go przeprowadzono, i od wersji całego użytego wtedy oprogramowania, mogą i będą się one zmieniać z upływem czasu. Wynika to stąd, że nowe urządzenia się różnią, nowe wersje każdego pakietu oprogramowania są inne, a także zmieniają się charakterystyki wydajnościowe różnych kombinacji sprzętu i (lub) oprogramowania. Ponadto ustawienia konfiguracyjne zastosowane w teście porównawczym mają znaczny wpływ na uzyskiwane wyniki. Gdy Czytelnik będzie to czytał, prawdopodobnie poniższe rezultaty będą zdezaktualizowane. Jeśli nawet książka zostanie przeczytana wkrótce po wprowadzeniu jej do sprzedaży, zastosowane przez Czytelnika urządzenia i oprogramowanie raczej nie będą takie same jak wykorzystane przez nas. Jedyną metodą pozwalającą faktycznie stwierdzić, jak na komputerze będzie działać serwer Tomcat i (lub) serwer *httpd* Apache, jest przeprowadzenie we własnym zakresie testu porównawczego zgodnie z poniższą procedurą testową.

Złącza Tomcata i moduły złącza serwera httpd Apache

Serwer Tomcat oferuje implementacje trzech różnych rozwiązań serwerowych obsługujących żądania HTTP, a także implementacje tych samych trzech mechanizmów przetwarzających żądania AJP.

JIO (`java.io`)

Jest to domyślna implementacja złącza serwera Tomcat, jeśli podczas jego ładowania nie zostanie znaleziona biblioteka *libtcnative* elementu `Connector` biblioteki APR. Implementacja jest też znana pod nazwą Coyote. Ta serwerowa implementacja gniazd TCP utworzona w czystym języku Java korzysta z podstawowych klas sieciowych Java o nazwie `java.io`. Jest to implementacja zarówno protokołu HTTP, jak i AJP z funkcją pełnego blokowania. Fakt, że napisano ją w czystym języku Java, sprawia, że na poziomie binarnym można ją przenieść do wszystkich systemów operacyjnych w pełni obsługujących środowisko Java. Wiele osób jest przekonanych, że ta implementacja będzie wolniejsza od serwera *httpd* Apache, głównie dlatego, że utworzono ją w języku Java. Przyjmuje się, że aplikacja Java zawsze jest wolniejsza od skompilowanej w języku C. Czy tak naprawdę jest? Sprawdzimy to.

APR (Apache Portable Runtime)

Jest to domyślna implementacja złącza serwera Tomcat, gdy zainstaluje się go w systemie Windows za pomocą narzędzia NSIS. Jednak nie jest tak w przypadku większości innych instalacji Tomcata. Implementacja ta ma postać kilku klas Java uwzględniających składnik JNI, opakowujący niewielką bibliotekę *libtcnative*, napisaną w języku programowania C, która jest zależna od biblioteki APR. Serwer WWW *httpd* Apache również jest implementowany w języku C i na potrzeby własnej komunikacji sieciowej wykorzystuje bibliotekę ARP. Jednym z zadań tej alternatywnej implementacji jest zapewnienie serwerowego rozwiązania używającego tego samego udostępnionego kodu źródłowego języka C co serwer *httpd* Apache. Ma to na celu zdystansowanie złącza JIO, a także zaoferowanie wydajności, która przynajmniej dorównuje osiągom serwera *httpd* Apache. Wadą implementacji APR jest to, że ponieważ przede wszystkim napisano ją w języku C, jedna binarna wersja tego elementu `Connector` nie zostanie zastosowana w przypadku wszystkich platform obsługiwanych przez złącze JIO. Oznacza to, że administratorzy Tomcata muszą budować złącze APR. W związku z tym wymagane jest środowisko programowania, a ponadto mogą wystąpić problemy z procesem budowania. Jednak twórcy tej wersji elementu `Connector` uzasadniają dodatkowe czynności przygotowawcze tym, że wydajność serwera WWW Tomcata będzie większa, gdy się ją zastosuje. Sami się o tym przekonamy, przeprowadzając testy porównawcze.

NIO (`java.nio`)

Jest to alternatywna implementacja elementu `Connector` napisana w czystym języku Java, która używa podstawowych klas sieciowych Java o nazwie `java.nio`, oferujących funkcje gniazd TCP bez blokowania. Głównym zadaniem tej implementacji elementu `Connector` jest zapewnienie administratorom Tomcata rozwiązania, które funkcjonuje efektywniej od złącza JIO. W przypadku tej implementacji użyto mniejszej liczby wątków przez zrezygnowanie z blokowania dla wybranych składników złącza. Fakt blokowania przez złącze JIO odczytów i zapisów oznacza, że jeśli administrator skonfiguruje złącze pod kątem jednoczesnej obsługi 400 połączeń, będzie ono musiało wywołać 400 wątków środowiska Java. Z kolei złącze NIO wymaga tylko jednego wątku do przetwarzania żądań wielu połączeń. Jednak później każde żądanie skierowane do serwletu musi dysponować własnym wątkiem (ograniczenie narzucone przez specyfikację Java Servlet Specification). Ponieważ część obsługi żądania ma miejsce w kodzie Java bez blokowania, w czasie wymaganym na zrealizowanie tego zadania wątek Java nie musi być używany. Oznacza to, że do obsługi tej samej liczby jednoczesnych żądań może być wykorzystana mniejsza pula wątków. Zwykle wiąże się z tym mniejsze obciążenie procesora, a to z kolei powoduje wzrost wydajności. Teoria wyjaśniająca, dlaczego coś takiego przyczyni się do zwiększenia wydajności, opiera się na sporym zestawie założeń, które mogą (lub nie) dotyczyć aplikacji WWW i ruchu sieciowego generowanego przez dowolną osobę. Dla części osób złącze NIO będzie działać lepiej, a dla części gorzej (tak też jest w przypadku innych implementacji złączy).

Oprócz tych złączy serwera Tomcat przetestowaliśmy serwer *httpd* Apache w konfiguracjach Prefork i Worker modelu MPM (*Multi-Process Model*), w przypadku których testowe żądania były wysyłane z serwera *httpd* Apache do serwera Tomcata za pośrednictwem modułu złącza tego pierwszego. Test porównawczy wykonaliśmy dla następujących modułów złączy serwera *httpd* Apache:

mod_jk

Moduł ten jest rozwijany w ramach projektu serwera Apache Tomcat. Projekt ten rozpoczęto wiele lat przed dodaniem do modułu *mod_proxy* serwera *httpd* Apache obsługi protokołu AJP (złącza AJP Tomcata implementują protokół po stronie serwera). Moduł *mod_jk* jest modułem serwera *httpd* Apache, który implementuje protokół AJP po stronie klienta. Jest to binarny protokół bazujący na pakietach TCP, którego zadaniem jest przekazywanie zawartości żądań HTTP do instancji innego serwera znacznie szybciej niż byłoby to możliwe w przypadku samego protokołu HTTP. Założenie jest takie, że protokół HTTP jest w dużym stopniu ukierunkowany na przesyłanie zwykłego tekstu, a zatem po serwerowej stronie połączenia wymaga wolniejszych i bardziej złożonych analizatorów. Jeśli zamiast protokołu HTTP zastosuje się binarny protokół przekazujący poddane już analizie łańcuchy tekstowe żądań, serwer może odpowiedzieć znacznie szybciej, a ponadto może być zminimalizowane obciążenie wywołane przez komunikację sieciową. Taka przynajmniej jest teoria. Przekonamy się, jak bardzo różni się to w praktyce. Gdy pisano książkę, większość użytkowników serwera *httpd* Apache dodających Tomcata do swoich serwerów WWW w celu obsługi serwletów i (lub) stron JSP budowała moduł *mod_jk* i korzystała z niego głównie dlatego, że była przekonana, że jest on znacznie szybszy od modułu *mod_proxy*, bądź dlatego, że nie miała świadomości, że moduł *mod_proxy* jest prostszą alternatywą. Innym powodem takiego stanu rzeczy było to, że ktoś zaproponował takim użytkownikom zastosowanie modułu *mod_jk*. Postanowiliśmy sprawdzić, czy budowanie, instalowanie, konfigurowanie i utrzymywanie modułu *mod_jk* jest warte uzyskanej wydajności.

mod_proxy_ajp

Jest to odmiana modułu *mod_proxy*, która obsługuje złącze protokołu AJP. Moduł *mod_*
↪*proxy_ajp* łączy się z portem protokołu AJP serwera Tomcat za pośrednictwem protokołu TCP, wysyła żądania do Tomcata, oczekuje na jego odpowiedzi, a następnie serwer *httpd* Apache przekazuje je do klientów WWW. Żądania w dwóch kierunkach są przesyłane między serwerami *httpd* Apache i Tomcat. Tak jak w przypadku modułu *mod_jk* między tymi serwerami pośredniczy protokół AJP. Złącze tego protokołu stało się częścią serwera *httpd* Apache, począwszy od wersji 2.2, i aktualnie wchodzi w skład serwera Apache dołączanego do większości systemów operacyjnych lub jest domyślnie dodawane w postaci ładowalnego modułu serwera *httpd*. W celu użycia złącza zwykle nie jest konieczna żadna dodatkowa kompilacja lub instalacja. Wystarczy skonfigurować serwer *httpd* Apache. Ponieważ moduł *mod_proxy_ajp* bazuje na module *mod_jk*, kod i funkcje obu tych modułów są do siebie bardzo podobne.

mod_proxy_http

Jest to odmiana modułu *mod_proxy*, która obsługuje złącze protokołu HTTP. Podobnie do modułu *mod_proxy_ajp*, moduł ten łączy się z Tomcatem za pośrednictwem protokołu TCP, lecz w celu nawiązania połączenia z portem HTTP serwera WWW Tomcata. Oto proste przedstawienie sposobu działania modułu *mod_proxy_http* — klient WWW wysyła żądanie do serwera *httpd* Apache, a ten następnie to samo żądanie kieruje do serwera WWW Tomcata, który odpowiada serwerowi *httpd*, a ten odpowiedź przekazuje klientowi WWW. Gdy używa się modułu *mod_proxy_http*, cała komunikacja między serwerami *httpd* Apache i Tomcat jest realizowana za pomocą protokołu HTTP. Również moduł złącza protokołu HTTP wchodzi w skład serwera *httpd* Apache. Zazwyczaj moduł *mod_proxy_http* jest częścią binariów serwera *httpd*, dołączonych do większości systemów operacyjnych. Ponieważ moduł od bardzo dawna jest elementem serwera *httpd* Apache, jest dostępny niezależnie od jego zastosowanej wersji.

Konfiguracje sprzętowe i programowe wykorzystane w testach porównawczych

Do testowania działającego oprogramowania wybraliśmy dwie różnego typu serwerowe platformy sprzętowe. Poniżej zamieszczono opis dwóch komputerów, które posłużyły do przeprowadzenia testów porównawczych.

Komputer stacjonarny: dwa procesory Intel Xeon 64 2,8 GHz, pamięć RAM 4 GB, dysk twardy SATA 160 GB o prędkości 7200 obrotów na minutę.

Jest to komputer w obudowie typu *Tower* z dwoma 64-bitowymi procesorami Intela, z których każdy wyposażono w jeden rdzeń i technologię hiperwątkowości.

Laptop: procesor AMD Turion64 ML-40 2,2 GHz, pamięć RAM 2 GB, dysk twardy IDE 80 GB o prędkości 5400 obrotów na minutę.

Jest to laptop z pojedynczym 64-bitowym jednordzeniowym procesorem AMD.

Ponieważ jeden z komputerów jest stacjonarny, a drugi to laptop, wyniki testów pokażą również różnice w możliwości udostępniania statycznych plików przez jedno- i dwuprocesorowy komputer. Nie próbowaliśmy dopasowywać dwóch odmiennych modeli procesorów pod względem podobnej mocy obliczeniowej, lecz testom porównawczym poddaliśmy typowy 2-procesorowy komputer stacjonarny i 1-procesorowy laptop, które w czasie przeprowadzania testów były nowością na rynku. Oba komputery na dyskach twardych mają zwykłe partycje `ext3`. Z tego względu w testach porównawczych nie wykorzystano żadnych konfiguracji LVM lub RAID.

Choć oba komputery bazują na architekturze x86_64, ich procesory zostały zaprojektowane i wyprodukowane przez różne firmy. Ponadto komputery te wyposażono w kartę sieciową Gigabit Ethernet i testowano je z innego komputera, również dysponującego taką samą kartą. Komputery podłączono do przełącznika sieciowego obsługującego technologię Gigabit Ethernet.

Zdecydowaliśmy się zastosować klienta testującego ApacheBench (ab). Zależało nam na zapewnieniu obsługi przez klienta funkcji ciągle aktywnych połączeń HTTP 1.1, ponieważ właśnie ją chcieliśmy poddać testom. Poza tym oczekiwaliśmy, że klient będzie na tyle szybki, że wygeneruje jak najbardziej dokładne wyniki. Zapoznaliśmy się z artykułem zamieszczonym na blogu przez Scotta Oaksa (*http://weblogs.java.net/blog/sdo/archive/2007/03/ab_considered_h.html*). Choć zgadzamy się z analizą działania narzędzia ab dokonaną przez Oaksa, dokładnie monitorowaliśmy obciążenie procesora przez klienta testującego ab i zyskaliśmy pewność, że w czasie wykonywanych testów nigdy nie przeciążył wykorzystywanego procesora. Dodatkowo dla narzędzia ab uaktywniliśmy funkcję, która pozwala na jednoczesną aktywność więcej niż jednego żądania HTTP. Fakt, że pojedynczy proces narzędzia ab może używać dokładnie jednego procesora nie stanowi problemu, gdyż system operacyjny przełącza konteksty w procesorze szybciej, niż sieć jest w stanie wysyłać i odbierać żądania, a także pakiety odpowiedzi. Okazuje się, że dla procesora w rzeczywistości wszystko jest pojedynczym strumieniem instrukcji sprzętowych. Ponieważ podczas przeprowadzanych przez nas testów komputer serwera WWW nie dysponował wystarczającą liczbą rdzeni procesora, żeby przeciążyć procesor komputera z uruchomionym narzędziem ab, w rzeczywistości mierzyliśmy wydajność samego serwera WWW.

Testom poddaliśmy serwer Tomcat 6.0.1 (gdy rozpoczynaliśmy testy, była to najnowsza dostępna wersja; spodziewamy się, że nowsze wersje będą szybsze, lecz do momentu wykonania testu nigdy nie można być tego pewnym) załadowany w środowisku Sun Java 1.6.0 GA przeznaczonym dla platformy x86_64. Dodatkowo testowaliśmy serwer Apache 2.2.3, moduł

mod_jk wchodzący w skład oprogramowania Tomcat Connectors (wersja 1.2.20) i złącze biblioteki APR (*libtcnative*) w wersji 1.1.6. W czasie przeprowadzania testów były to najnowsze wersje oprogramowania. Przykro nam, że na potrzeby książki nie mogliśmy przetestować nowszych wersji. Jednak znakomitą rzeczą związaną ze szczegółowo opisanymi testami jest to, że zapewniają ilość informacji wystarczającą do powtórzenia ich we własnym zakresie. Na obu komputerach zainstalowano system operacyjny Fedora Core 6 Linux x86_64 z aktualizacjami dokonanymi za pomocą narzędzia `yum`. Użyto jądra w wersji 2.6.18.2.

Oto zastosowane ustawienia startowych parametrów wirtualnej maszyny Java serwera Tomcat:

```
-Xms384M -Xmx384M -Djava.awt.headless=true -Djava.net.preferIPv4Stack=true
```

Konfiguracja serwera Tomcat wykorzystana na potrzeby testów uwzględnia plik *conf/web.xml*, plik *conf/server.xml* (z tym że nie włączono rejestratora prób dostępu; w efekcie nie będą rejestrowane informacje dla poszczególnych żądań), a także poniższe konfiguracje złączy, z których dla różnych testów w danej chwili było stosowane tylko jedno.

```
<!-- Złącze JIO HTTP -->
<Connector port="8080" protocol="HTTP/1.1"
           maxThreads="150" connectionTimeout="20000"
           redirectPort="8443" />

<!-- Złącze APR HTTP -->
<Connector port="8080"
           protocol="org.apache.coyote.http11.Http11AprProtocol"
           enableLookups="false" redirectPort="8443"
           connectionTimeout="20000"/>

<!-- Złącze NIO HTTP -->
<Connector port="8080"
   maxThreads="150" connectionTimeout="20000"
   redirectPort="8443"
   protocol="org.apache.coyote.http11.Http11NioProtocol"/>

<!-- Złącze JIO/APR AJP modyfikowane przez ustawienie LD_LIBRARY_PATH -->
<Connector port="8009" protocol="AJP/1.3" redirectPort="8443" />

<!-- Złącze NIO AJP -->
<Connector protocol="AJP/1.3" port="0"
      channelNioSocket.port="8009"
      channelNioSocket.maxThreads="150"
      channelNioSocket.maxSpareThreads="50"
      channelNioSocket.minSpareThreads="25"
      channelNioSocket.bufferSize="16384"/>
```

Kod biblioteki APR został uaktywniony za pomocą przedstawionej konfiguracji złącza APR HTTP, a także przez ustawienie katalogu zawierającego bibliotekę *libtcnative* dla zmiennej środowiskowej `LD_LIBRARY_PATH` procesu wirtualnej maszyny Java Tomcata i wyeksportowanie jej, a następnie zrestartowanie serwera Tomcat.

Złącze APR jest budowane w następujący sposób:

```
# CFLAGS="-O3 -falign-functions=0 -march=athlon64 -mfpmath=sse -mmmx -msse -msse2 -
msse3 -m3dnow -mtune=athlon64" ./configure --with-apr=/usr/bin/apr-1-config --
prefix=/opt/tomcat/apr-connector
# make && make install
```

Przeprowadzając proces budowania serwera *httpd* Apache i modułu *mod_jk*, zastosowaliśmy takie same zmienne środowiskowe `CFLAGS`. Moduł *mod_jk* zbudowaliśmy i zainstalowaliśmy w następujący sposób:

```
# cd tomcat-connectors-1.2.20-src/native
# CFLAGS="-O3 -falign-functions=0 -march=athlon64 -mfpmath=sse -mmmx -msse -msse2 -
msse3 -m3dnow -mtune=athlon64" ./configure --with-apxs=/opt/httpd/bin/apxs
[usunięto mnóstwo wynikowych danych konfiguracyjnych]
# make && make install
```

Przyjęto, że główny katalog zbudowanego serwera *httpd* Apache identyfikuje ścieżka */opt/httpd*.

Proces budowania złącza APR, serwera *httpd* i modułu *mod_jk* zrealizowano za pomocą narzędzia GCC 4.1.1.

```
# gcc --version
gcc (GCC) 4.1.1 20061011 (Red Hat 4.1.1-30)
Copyright (C) 2006 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

Z witryny o adresie *http://httpd.apache.org* pobraliśmy serwer *httpd* Apache w wersji 2.2.3, a następnie zbudowaliśmy go na dwa różne sposoby i przetestowaliśmy każde z uzyskanych binariów. Proces budowania został przeprowadzony dla konfiguracji Prefork i Worker modelu MPM. Są to różne odmiany modelu wielowątkowego i wieloprocesowego, z których serwer może korzystać. Oto ustawienia użyte dla konfiguracji Prefork i Worker modelu MPM:

```
# Prefork MPM
<IfModule prefork.c>
StartServers       8
MinSpareServers    5
MaxSpareServers   20
ServerLimit      256
MaxClients       256
MaxRequestsPerChild  4000
</IfModule>

# Worker MPM
<IfModule worker.c>
StartServers         3
MaxClients         192
MinSpareThreads      1
MaxSpareThreads     64
ThreadsPerChild     64
MaxRequestsPerChild  0
</IfModule>
```

Dla serwera *httpd* Apache wyłączyliśmy rejestrowanie zwykłego dostępu, żeby w dzienniku nie musiało być nic zapisywane dla każdego żądania (tak samo skonfigurowaliśmy Tomcata). Uaktywniliśmy opcję konfiguracyjną `KeepAlive` serwera *httpd* Apache.

```
KeepAlive On
MaxKeepAliveRequests 100
KeepAliveTimeout 5
```

Przy użyciu jednej z dwóch metod uaktywniliśmy moduł *mod_proxy*. Najpierw włączyliśmy usługę proxy za pośrednictwem protokołu HTTP.

Oto wiersze uaktywniające usługę proxy za pośrednictwem protokołu AJP:

```
ProxyPass        /tc ajp://127.0.0.1:8009/
ProxyPassReverse /tc ajp://127.0.0.1:8009/
```

Skonfigurowaliśmy moduł *mod_jk* przez dodanie do pliku *httpd.conf* następujących wierszy:

```
LoadModule    jk_module  /opt/httpd/modules/mod_jk.so
JkWorkersFile /opt/httpd/conf/workers.properties
JkLogFile     /opt/httpd/logs/mod_jk.log
JkLogLevel    info
```

```
JkLogStampFormat "[%a %b %d %H:%M:%S %Y] "
JkOptions     +ForwardKeySize +ForwardURICompat -ForwardDirectories
JkRequestLogFormat     "%w %V %T"
JkMount  /tc/* worker1
```

Dodatkowo dla modułu *mod_jk* utworzyliśmy plik *workers.properties* w ścieżce określonej w pliku *httpd.conf*.

```
worker.list=worker1
worker.worker1.type=ajp13
worker.worker1.host=localhost
worker.worker1.port=8009
worker.worker1.connection_pool_size=150
worker.worker1.connection_pool_timeout=600
worker.worker1.socket_keepalive=1
```

Oczywiście, w danej chwili w konfiguracji uaktywniliśmy tylko jeden moduł złącza serwera *httpd* Apache.

Procedura testu porównawczego

Przetestowaliśmy dwa różnego typu żądania dotyczące statycznych zasobów — niewielkich plików tekstowych i plików obrazów o rozmiarze 9 kB. W przypadku obu rodzajów testów porównawczych skonfigurowaliśmy serwer tak, żeby był w stanie jednocześnie obsłużyć co najmniej 150 połączeń klientów. Z kolei dla klienta testującego ustawiliśmy nie więcej niż 149 jednoczesnych połączeń, żeby nigdy nie podejmował próby użycia liczby połączeń przekraczającej możliwości serwera. Skonfigurowaliśmy klienta tak, żeby dla wszystkich testów stosował ciągle aktywne połączenia HTTP.

W przypadku niewielkich plików tekstowych testowaliśmy możliwości serwera dotyczące odczytywania żądania HTTP i zapisywania odpowiedzi na nie, gdy jej zawartość jest bardzo niewielka. Przede wszystkim testowano zdolność serwera do szybkiego odpowiadania podczas jednoczesnego przetwarzania wielu żądań. Ustawiliśmy klienta testującego w taki sposób, żeby zażądał pliku 100 000 razy z możliwością wykorzystania maksymalnie 149 jednoczesnych połączeń. Plik tekstowy utworzyliśmy w następujący sposób:

```
$ echo 'Witaj świecie.' > test.html
```

Plik ten skopiowaliśmy do katalogu *ROOT* serwera Tomcat, a także do głównego katalogu dokumentów serwera *httpd* Apache.

Poniżej zaprezentowano wiersz polecenia ab z argumentami zastosowanymi w testach porównawczych przeprowadzonych dla niewielkich plików tekstowych.

```
$ ab -k -n 100000 -c 149 http://192.168.1.2/test.html
```

Dla każdego testu odpowiednio zmodyfikowaliśmy adres URL żądania, żeby generowane żądania były kierowane do serwera, który miał być każdorazowo testowany.

W przypadku plików obrazów o rozmiarze 9 kB testowaliśmy zdolność serwera do udostępniania dużej ilości danych umieszczanych w zawartości odpowiedzi wysyłanej jednocześnie do wielu klientów. Ustawiliśmy klienta testującego w taki sposób, żeby zażądał pliku 20 000 razy z możliwością użycia maksymalnie 149 jednoczesnych połączeń. Dla tego testu określiliśmy mniejszą całkowitą liczbę żądań z powodu większego rozmiaru danych. Choć w związku z tym zredukowaliśmy odpowiednio liczbę żądań, w dalszym ciągu serwer jest przez nie w znaczącym stopniu obciążony. Oto polecenie tworzące plik obrazu:

```
$ dd if=a-wiekszy-obraz.jpg of=9k.jpg bs=1 count=9126
```

Wybraliśmy rozmiar 9 kB, gdyż w przypadku zastosowania większego serwery Tomcat i *httpd* Apache z łatwością przeciążyłyby używane przez nas łącze ethernetowe o szybkości 1 Mb znajdujące się między klientem i serwerem. Również ten plik skopiowaliśmy do katalogu *ROOT* serwera Tomcat i głównego katalogu dokumentów serwera *httpd* Apache.

Poniżej zamieszczono wiersz polecenia `ab` z argumentami użytymi w testach porównawczych przeprowadzonych dla plików obrazów.

```
$ ab -k -n 20000 -c 149 http://192.168.1.2/20k.jpg
```

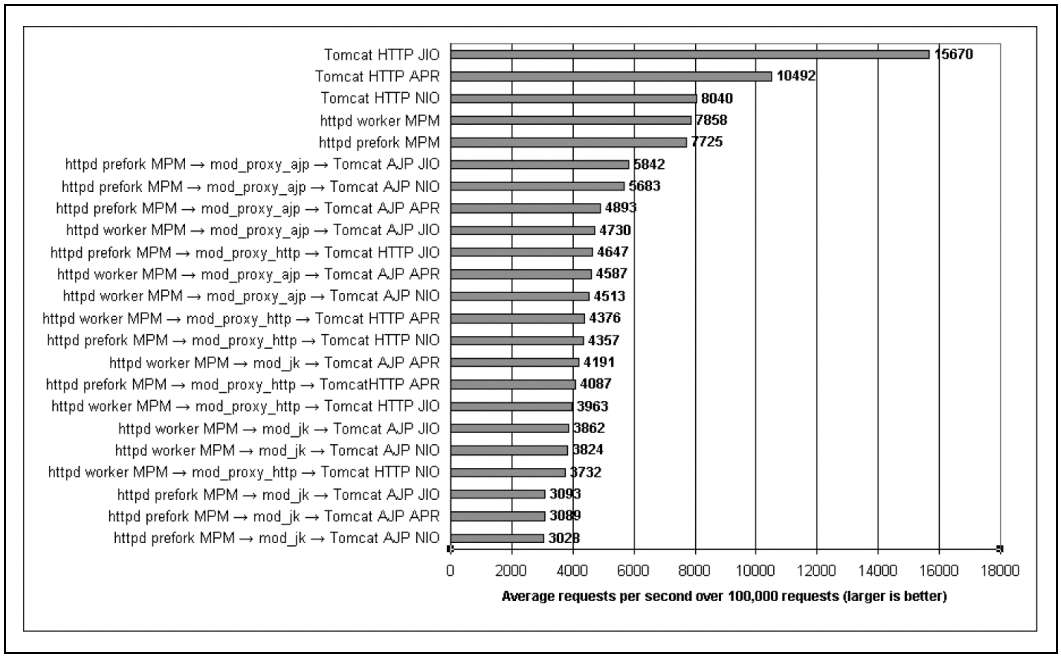
Dla każdego wywołania narzędzia ab uzyskaliśmy wyniki, postępując zgodnie z następującą procedurą:

1. Skonfigurować i zrestartować instancje testowanego serwera *httpd* Apache i (lub) serwera Tomcat.
2. Sprawdzić, czy serwer lub serwery nie zarejestrowały żadnych błędów podczas uruchamiania. Jeśli błędy pojawiły się, trzeba je usunąć przez kontynuowaniem.
3. Wywołać polecenie `ab`, żeby po zrestartowaniu serwery obsłużyły pierwsze żądania.
4. Ponownie wykonać polecenie `ab` w ramach testu porównawczego.
5. Po obsłużeniu wszystkich żądań upewnić się, że narzędzie ab poinformowało o braku błędów i odpowiedzi innych niż typu 2xx.
6. Między kolejnymi wywołaniami narzędzia ab odczekać kilka sekund, żeby serwery powróciły do stanu bezczynności.
7. W statystykach narzędzia ab zwrócić uwagę na liczbę żądań w ciągu sekundy.
8. Powrócić do kroku 4., jeśli znacząco zmienia się liczba żądań na sekundę. Gdy tak nie jest, uzyskana liczba żądań na sekundę będzie wynikiem testu porównawczego. Jeżeli liczby żądań w dalszym ciągu znacznie się wahają, testy należy zakończyć po 10 wywołaniach narzędzia ab i jako wynik zapisać ostatni odczyt liczby żądań na sekundę.

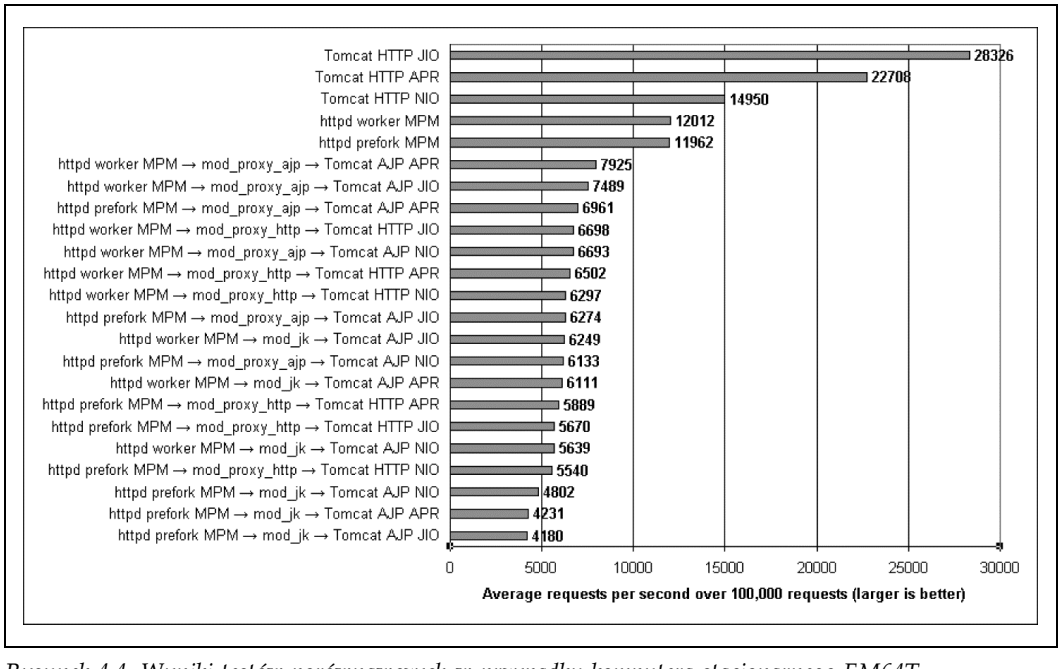
Rzecz w tym, że w przypadku kilku pierwszych wywołań narzędzia ab serwery będą nieefektywne. Jednak później oprogramowanie serwera uzyska stan, w którym wszystko będzie właściwie inicjalizowane. Wirtualna maszyna Java Tomcata rozpoczyna profilowanie i kompilowanie we własnym zakresie najczęściej stosowanego kodu w konkretnym przypadku użycia oprogramowania. Dzięki temu dodatkowo skraca się czas odpowiedzi. Kilka wywołań narzędzia ab jest potrzebnych do tego, żeby serwery uzyskały bardziej optymalny stan działania, który powinien być poddany testowi porównawczemu. Taki stan serwery osiągną, gdy tak jak serwery produkcyjne będą obsługiwały żądania od wielu godzin lub dni.

Wyniki testów porównawczych i podsumowanie

Przeprowadziliśmy testy porównawcze i dane wynikowe przedstawiliśmy na wykresie słupkowym. Konfiguracje serwerów WWW zestawiono zgodnie z malejącą wydajnością (jeden wykres przypada na pojedynczy test wykonany na jednym komputerze). Najpierw sprawdzimy, jak wypadły komputery w teście porównawczym dotyczącym niewielkich plików tekstowych (rysunki 4.3 i 4.4).



Rysunek 4.3. Wyniki testów porównawczych w przypadku laptopa z 64-bitowym procesorem AMD udostępniającego niewielkie pliki tekstowe

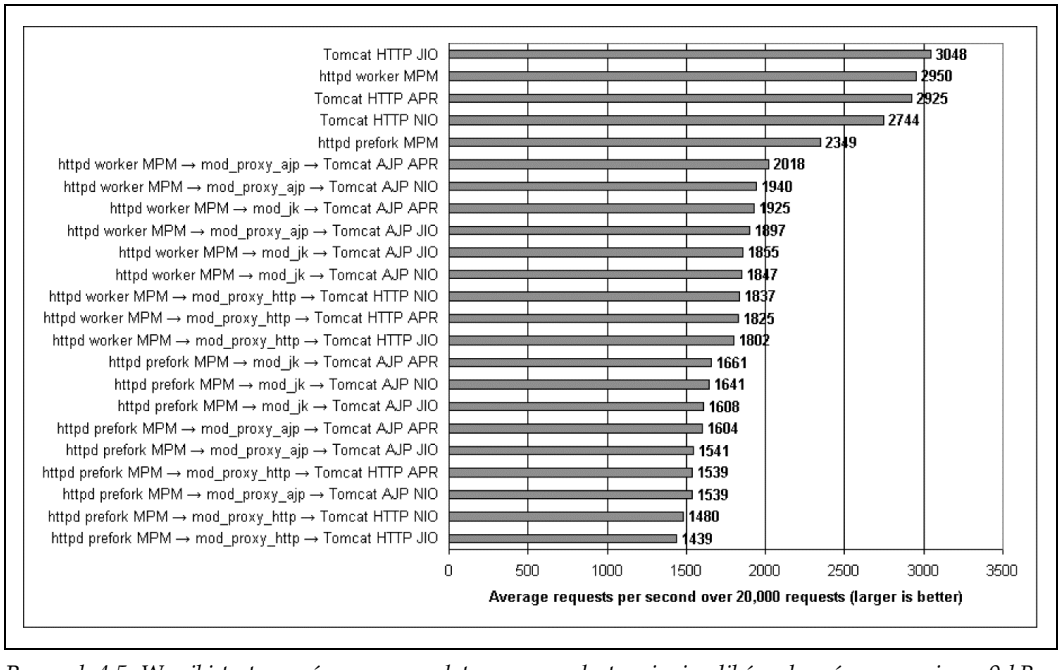


Rysunek 4.4. Wyniki testów porównawczych w przypadku komputera stacjonarnego EM64T udostępniającego niewielkie pliki tekstowe

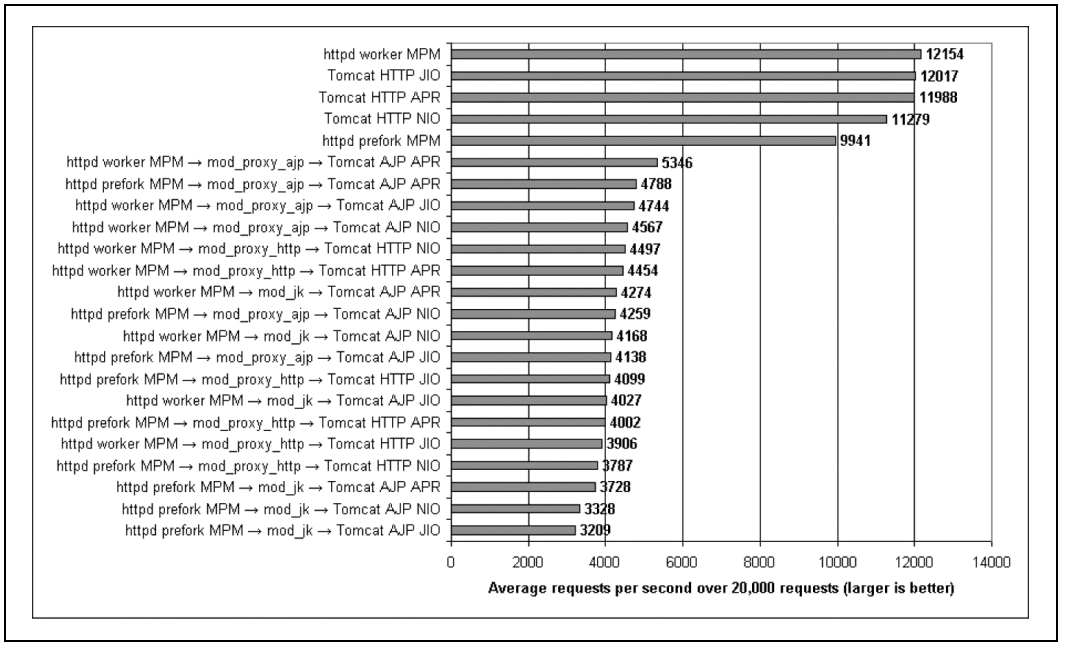
Warto zauważyć, że rysunki 4.3 i 4.4 wyglądają bardzo podobnie. W przypadku obu komputerów serwer Tomcat z niezależnym złączem JIO jest najszybszym serwerem WWW udostępniającym statyczne pliki tekstowe. Na dalszych pozycjach znalazły się serwery Tomcat ze złączami APR i NIO. Czwarte i piąte miejsce zajęły dwie konfiguracje serwera *httpd* Apache. Na dalszych lokatach umiejscowiły się wszystkie kombinacje serwera *httpd* Apache połączonego z Tomcatem za pośrednictwem modułu złącza. W dolnej części wykresu dominuje moduł *mod_jk*.

Interesujące jest również porównanie na obu wykresach uzyskanej liczby żądań na sekundę dla jednego serwera WWW. Laptop jest wyposażony w jednordzeniowy procesor, a komputer stacjonarny w dwa jednordzeniowe procesory. A zatem, gdy dwuprocesorowy komputer działa efektywnie, a także system operacyjny i wirtualna maszyna Java mogą skutecznie wykorzystać możliwości obu układów, komputer ten powinien być w stanie obsłużyć w ciągu sekundy prawie dwukrotnie większą liczbę żądań niż laptop z 64-bitowym procesorem AMD. Oczywiście, w tym przypadku przyjęto, że komputery dwuprocesorowe równie szybko wykonują instrukcje. Jednak tak może nie być. Gdy porówna się wyniki dotyczące dwóch komputerów, w przypadku komputera stacjonarnego serwer WWW o takiej samej konfiguracji w ciągu sekundy przetworzy niemal dwukrotnie większą liczbę żądań (trzeba uwzględnić obciążenie dwóch procesorów wynikające ze współużytkowania jednego zestawu zasobów systemowych, takich jak pamięć RAM, magistrale danych, magistrale wejścia-wyjścia itp.). Komputer z dwoma procesorami może obsłużyć prawie taką samą liczbę żądań co dwa komputery z pojedynczym procesorem. Skorzystać z tego może zarówno Tomcat, jak i serwer *httpd* Apache.

Następnie uzyskaliśmy wyniki testu porównawczego dotyczącego plików obrazów o rozmiarze 9 kB, przeprowadzonego na obu komputerach. Rysunki 4.5 i 4.6 prezentują rezultaty odpowiednio testu laptopa i komputera stacjonarnego.



Rysunek 4.5. Wyniki testu porównawczego dotyczącego udostępniania plików obrazów o rozmiarze 9 kB, wykonanego na laptopie



Rysunek 4.6. Wyniki testu porównawczego dotyczącego udostępniania plików obrazów o rozmiarze 9 kB, przeprowadzonego na komputerze stacjonarnym

Na rysunku 4.5 widać, że w przypadku laptopa ponownie zwycięzcą został serwer Tomcat z niezależnym złączem JIO, a tuż za nim uplasował się serwer *httpd* Apache w konfiguracji Worker modelu MPM. W tym teście porównawczym wydajność tych dwóch serwerów jest niemal identyczna. Niewiele gorzej wypadł serwer Tomcat z niezależnym złączem APR, który zajął trzecią lokatę. Na czwartym miejscu znalazł się Tomcat z niezależnym złączem NIO, osiągając wynik bardzo niewiele różniący się od wyniku z trzeciego miejsca. I tym razem w tyle za wszystkimi niezależnymi konfiguracjami serwera Tomcat znalazła się konfiguracja Prefork modelu MPM serwera *httpd* Apache, lokując się na piątym miejscu. W dalszym ciągu wolniejsze są wszystkie kombinacje serwera *httpd* Apache łączącego się z Tomcatem za pośrednictwem modułów złącza. Tym razem zaobserwowaliśmy, że moduł *mod_jk* uzyskał wynik mniej więcej na poziomie średniej wśród modułów złącza. Niektóre konfiguracje modułu *mod_proxy_http* wypadły najgorzej.

Rysunek 4.6 w pewnym sensie różni się, pokazując, że w przypadku 2-procesorowego komputera stacjonarnego serwer *httpd* Apache nieznacznie wyprzedza Tomcata z najszybszym niezależnym złączem JIO. Różnica w wydajności między tymi dwoma serwerami jest znikoma i wynosi około 1 procenta. Może to wskazywać na to, że istnieje różnica w sposobie funkcjonowania komputera stacjonarnego i laptopa. Okazuje się, że serwer *httpd* Apache jest o 1 procent szybszy od Tomcata w przypadku komputera stacjonarnego udostępniającego pliki obrazów (przynajmniej dotyczy to komputerów użytych przez nas w testach). Nie powinno się zakładać, że tak będzie w przypadku nowszych komputerów, ponieważ zmianie ulega wiele szczegółów dotyczących sprzętu! Ponadto zauważyliśmy, że w przeprowadzonym zestawie testów wszystkie trzy niezależne złącza serwera Tomcat wypadły lepiej od konfiguracji Prefork modelu MPM serwera *httpd* Apache. Konfiguracje, w których serwer *httpd* Apache łączy się z Tomcatem za pośrednictwem modułu złącza, ponownie cechowały się najgorszą wydajnością. W tej grupie najwolniejszy okazał się moduł *mod_jk*.

Czy w przypadku plików obrazów komputer stacjonarny z dwoma procesorami ponownie w ciągu sekundy w przybliżeniu będzie obsługiwał dwukrotnie więcej żądań od laptopa z jednym procesorem? Odpowiedź jest przecząca. Z jakiegoś powodu wynik jest lepszy i wynosi czterokrotnie większą liczbę żądań przetworzonych na sekundę. Jak to może być możliwe, że po zainstalowaniu dodatkowego procesora komputer może pracować cztery razy szybciej? Prawdopodobnie tak nie jest. Jedynym wyjaśnieniem, jakie przychodzi nam na myśl, jest to, że coś powoduje, że w przypadku udostępniania plików obrazów laptop nie może wykorzystać pełnego potencjału procesora. Niekonieczne jest to spowodowane przez problem ze sprzętem. Może to wynikać z tego, że sterownik urządzenia nie działa efektywnie w przypadku konkretnej wersji jądra i obniża wyniki testu porównawczego. To wskazuje na to, że ze względu na wolny sterownik mogą nie być dokładne wyniki testów dotyczące plików obrazów o rozmiarze 9 kB, przeprowadzone na laptopie. Jednak taką wydajność zaobserwowano tylko na tym komputerze. Tak będzie działał, dopóki i jeśli inna wersja jądra nie poprawi wydajności komputera. Wiedząc o tym, nie jest jasne, czy pliki obrazów są udostępniane szybciej przez Tomcata, czy przez serwer *httpd* Apache. Niezależnie od tego możemy przypuszczać, że bardziej precyzyjne są wyniki testów porównawczych uzyskane w przypadku komputera stacjonarnego.

Oto podsumowanie uzyskanych wyników testów porównawczych z uwzględnieniem kilku istotnych statystyk:

- We wszystkich naszych testach porównawczych, z wyjątkiem testu dotyczącego plików obrazów o rozmiarze 9 kB i wykonanego na komputerze z 64-bitowymi procesorami Intel Xeon, niezależny serwer Tomcat okazał się szybszy od serwera *httpd* Apache skompilowanego w konfiguracji Worker modelu MPM. Nawet w przypadku tego jednego testu serwer *httpd* był zaledwie o 1 procent szybszy od Tomcata. Zaobserwowaliśmy, że serwer Tomcat z niezależnym złączem JIO prawie zawsze był najszybszą metodą udostępniania statycznych zasobów. W naszych testach Tomcat oferował je od 3 do 136 procent szybciej od serwera *httpd* Apache. W przypadku plików obrazów o rozmiarze 9 kB niezależne złącze JIO serwera Tomcat okazało się co najmniej o 3 procent szybsze od konfiguracji Worker modelu MPM serwera *httpd* Apache. Wyjątkiem był test porównawczy przeprowadzony na komputerze z 64-bitowymi procesorami Intel Xeon, w którego przypadku serwer *httpd* okazał się o 1 procent szybszy od Tomcata. Jednak w teście porównawczym dotyczącym niewielkich plików tekstowych serwer Tomcat był o co najmniej 99 procent bardziej wydajny od serwera *httpd* Apache i maksymalnie szybszy od niego o 136 procent.
- Konfiguracja Worker modelu MPM serwera *httpd* Apache okazała się najszybsza wśród wszystkich jego testowanych konfiguracji. We wszystkich testach konfiguracja Prefork modelu MPM serwera *httpd* była wolniejsza od konfiguracji Worker. Zauważyliśmy, że konfiguracja Worker udostępnia dane o co najmniej 0,4 procent szybciej od konfiguracji Prefork modelu MPM i maksymalnie o 26 procent efektywniej od niej. Choć w przypadku testów porównawczych związanych z niewielkimi plikami tekstowymi między tymi dwoma konfiguracjami prawie w ogóle nie było różnicy w wydajności, przy testach dotyczących plików obrazów o rozmiarze 9 kB różnica wyniosła przynajmniej 22 procent.
- Niezależny serwer Tomcat (skonfigurowany pod kątem użycia dowolnej implementacji złącza HTTP) zawsze był szybszy niż serwer *httpd* Apache w konfiguracji Prefork modelu MPM. W tym przypadku w testach dotyczących plików obrazów o rozmiarze 9 kB Tomcat okazał się o co najmniej 21 procent wydajniejszy od serwera *httpd* Apache i maksymalnie o 30 procent szybszy od niego. Jeśli chodzi o udostępnianie niewielkich plików tekstowych, serwer Tomcat był przynajmniej o 103 procent szybszy od serwera *httpd* Apache w konfiguracji Prefork modelu MPM i maksymalnie o 136 procent efektywniejszy od niego.

- Serwer *httpd* Apache znacząco wolniej udostępniał niewielkie pliki. Tomcat z niezależnymi złączami JIO, APR i NIO w każdym przypadku wypadł lepiej niż serwer *httpd* Apache. Złącze JIO Tomcata było o 136 procent szybsze od najbardziej wydajnej konfiguracji serwera *httpd* Apache. Złącze APR serwera Tomcat działało o 89 procent szybciej od serwera *httpd* Apache, a złącze NIO Tomcata 6.0 okazało się o 25 procent wydajniejsze niż serwer *httpd*. W przypadku tego testu porównawczego serwer *httpd* Apache spadł na czwarte miejsce, lokując się za wszystkimi trzema niezależnymi złączami HTTP serwera Tomcat.
- Udostępnianie zasobów Tomcata za pośrednictwem serwera *httpd* Apache przebiegało bardzo wolno w porównaniu z bezpośrednim uzyskiwaniem dostępu do nich. Jeżeli porównamy wyniki testów uzyskane w przypadku niezależnego serwera Tomcat i Tomcata oferującego zasoby za pośrednictwem serwera *httpd* Apache korzystającego z modułu *mod_proxy*, okaże się, że samodzielnie działający Tomcat udostępniał dane co najmniej o 51 procent szybciej od serwera *httpd* Apache (z uwzględnieniem jego wszystkich trzech modułów złącza: *mod_jk*, *mod_proxy_ajp* i *mod_proxy_http*), gdy korzystał tylko ze swojego złącza JIO. W teście porównawczym dotyczącym niewielkich plików tekstowych niezależny serwer Tomcat był przynajmniej o 168 procent szybszy od konfiguracji, w których w dostępie do zasobów Tomcata pośredniczył serwer *httpd* Apache, a maksymalnie różnica w wydajności wyniosła 578 procent! To nie jest błąd w druku. Naprawdę Tomcat był o 578 procent szybszy. W przypadku testu porównawczego związanego z plikami obrazów o rozmiarze 9 kB niezależny serwer Tomcat był co najmniej o 51 procent i maksymalnie o 274 procent wydajniejszy.
- Protokół AJP zdystansował protokół HTTP, gdy zastosowano moduł *mod_proxy*. Wyniki testów pokazują, że moduł *mod_proxy_ajp* zawsze był szybszy od modułu *mod_proxy_http*. Choć w przypadku korzystania z tego samego złącza serwera Tomcat różnica w szybkości między tymi dwoma protokołami zawierała się w przedziale od 1 do 30 procent, zwykle wydajność była mniejsza od maksymalnej. Moduł *mod_proxy_ajp* średnio był szybszy od modułu *mod_proxy_http* o mniej więcej 13 procent.
- Udostępnianie statycznych zasobów Tomcata za pośrednictwem modułu złącza serwera *httpd* Apache nigdy nie było szybsze od oferowania tych samych zasobów przy wykorzystaniu tylko samego serwera *httpd* Apache. Wyniki testów porównawczych dotyczących udostępniania zasobów Tomcata za pomocą modułu złącza serwera *httpd* zawsze były gorsze niż w przypadku oferowania zasobów bezpośrednio przez serwer *httpd* Apache. Oznacza to, że test porównawczy przeprowadzony dla niezależnego serwera *httpd* Apache da wynik nieznacznie lepszy od teoretycznego maksimum, które można by uzyskać przy udostępnianiu tych samych zasobów za pośrednictwem modułu złącza serwera *httpd*. Ponadto wynika z tego, że niezależnie od tego, jak wydajny będzie serwer Tomcat, udostępnianie jego plików za pośrednictwem serwera *httpd* Apache spowoduje taki spadek efektywności Tomcata, że stanie się wolniejszy od serwera *httpd*.
- Moduł *mod_jk* nie był szybszy od modułu *mod_proxy*, z wyjątkiem testu dotyczącego plików obrazów o rozmiarze 9 kB i tylko w przypadku laptopa. Po przeprowadzonych przez nas testach okazało się, że udostępnianie zasobów Tomcata za pośrednictwem serwera *httpd* Apache stosującego moduł *mod_jk* było szybsze od wariantu z wykorzystaniem modułu *mod_proxy* tylko w przypadku laptopa i testów porównawczych powiązanych z plikami obrazów o rozmiarze 9 kB. We wszystkich innych testach moduł *mod_jk* był wolniejszy od modułu *mod_proxy_ajp*.

Jak to możliwe, że napisany w czystym języku Java serwer Tomcat udostępnia statyczne zasoby szybciej niż serwer *httpd* Apache? Podstawowa przyczyna tego stanu rzeczy, która przychodzi nam na myśl, jest taka, że ze względu na to, że Tomcat stworzono w języku Java i jego kod bajtowy może być skompilowany i dobrze zoptymalizowany podczas uruchamiania oprogramowania, poprawnie napisany kod Java może być wyjątkowo szybki, gdy wywoła się go w dopracowanej wirtualnej maszynie Java, implementującej wiele funkcji optymalizujących na etapie uruchamiania (na przykład Sun HotSpot JVM). Gdy kod przetworzy wiele żądań, wirtualna maszyna Java będzie „wiedziała", jak zoptymalizować go pod kątem konkretnego zastosowania w przypadku określonej konfiguracji sprzętowej. Z kolei serwer *httpd* Apache napisano w języku C, którego kod przed uruchomieniem jest całkowicie kompilowany. Jeśli nawet nakaże się kompilatorowi przeprowadzenie intensywnej optymalizacji binariów, nie będzie miała miejsca żadna optymalizacja na etapie wywoływania oprogramowania. A zatem nie jest możliwe, żeby serwer *httpd* Apache skorzystał z wielu optymalizacji fazy uruchomieniowej, które są charakterystyczne dla serwera Tomcat.

Innym potencjalnym powodem, dla którego Tomcat wydajniej udostępnia dane niż serwer *httpd* Apache, jest to, że każda wersja wirtualnej maszyny Java firmy Sun wydaje się szybciej wykonywać kod Java. Dotyczy to wielu cyklów wprowadzania wersji wirtualnej maszyny Java tego producenta. Oznacza to, że jeśli nawet nie modyfikuje się często programu Java, żeby wydajniej działał, prawdopodobnie jego efektywność będzie się poprawiać za każdym razem, gdy załaduje się go za pomocą nowszej i szybszej wirtualnej maszyny Java (pod warunkiem, że w takim samym tempie w dalszym ciągu będzie zwiększała się jej wydajność). W związku z tym dodatkowo pojawia się założenie, że nowsze wirtualne maszyny Java będą wystarczająco zgodne, żeby bez żadnych modyfikacji można było uruchamiać kod bajtowy programów Java.

Co jeszcze mogliśmy przetestować?

W przeprowadzonym teście porównawczym sprawdziliśmy wydajność serwera WWW w przypadku udostępniania danych za pośrednictwem protokołu HTTP. Nie testowaliśmy wariantu z zastosowaniem protokołu HTTPS (protokół HTTP z szyfrowaniem). Charakterystyki wydajnościowe protokołów HTTP i HTTPS prawdopodobnie znacznie się różnią, gdyż w przypadku tego drugiego protokołu zarówno serwer, jak i klient muszą szyfrować i odszyfrowywać dane w obu kierunkach połączenia sieciowego. Obciążenie spowodowane przez szyfrowanie zmniejszy szybkość przetwarzania żądań i generowania odpowiedzi na nie (skala spadku wydajności będzie się zmieniać zależnie od różnych implementacji kodu szyfrującego). Nie testowaliśmy wydajności powyższych konfiguracji serwera WWW w przypadku użycia protokołu HTTPS. Nie przeprowadzając takiego testu, wiele osób jest przekonanych, że wydajność protokołu HTTPS serwera *httpd* Apache jest wyższa niż w przypadku serwera Tomcat. Zwykle osoby te bazują na tym, że kod języka C jest szybszy od kodu języka Java. W trzech z czterech scenariuszy wykonanych przez nas testów z zastosowaniem protokołu HTTP okazało się, że wcale tak nie jest. W przypadku czwartego scenariusza testów pod względem wydajności kod języka C wcale nie wypada znacznie lepiej. Bez przeprowadzenia testu porównawczego nie jesteśmy w stanie stwierdzić, która konfiguracja serwera WWW z protokołem HTTPS będzie najbardziej efektywna. Jeśli jednak kod szyfrujący języka C lub Java jest zdecydowanie najszybszy, Tomcat może użyć obu, ponieważ możliwe jest skonfigurowanie złącza APR w celu zastosowania bibliotek OpenSSL oferujących szyfrowanie HTTPS (z tych samych bibliotek języka C korzysta serwer *httpd* Apache).

Mogliśmy przetestować inne charakterystyki, takie jak przepustowość. Znacznie więcej interesujących rzeczy można się dowiedzieć przez obserwację dowolnej wybranej charakterystyki umieszczonej w raporcie przez narzędzie ab. Na potrzeby takiego testu większą wydajność określimy jako wyższą liczbę żądań z powodzeniem zrealizowanych w ciągu sekundy (kod odpowiedzi 2xx).

Mogliśmy poddać testom statyczne pliki o innych rozmiarach, w tym pliki o wielkości przekraczającej 9 kB. Jednak w przypadku plików o rozmiarze większym niż 100 kB wszystkie rozpatrywane konfiguracje serwerów spowodują przeciążenie megabitowego połączenia sieciowego Ethernet. W efekcie niemożliwe staje się zmierzenie, jak szybko samo oprogramowanie serwera może udostępnić pliki, gdyż połączenie sieciowe nie dysponowało wystarczającą przepustowością. Wykonując testy, do dyspozycji mieliśmy przepustowość nieprzekraczającą jednego megabitu.

Choć mogliśmy przeprowadzić testy dla plików o różnych wielkościach powiązanych z jednym żądaniem HTTP, jaki zestaw plików mielibyśmy wybrać i jaki przypadek użycia byłby przez niego reprezentowany? Wyniki tego typu testów porównawczych byłyby interesujące tylko wtedy, gdyby rzeczywiste obciążenie serwera WWW, którym się zarządza, było spowodowane przez wystarczająco podobny zestaw plików o różnych rozmiarach. Jest to mało prawdopodobne. W związku z tym skoncentrowaliśmy się na przeprowadzeniu testu porównawczego dla plików o dwóch rozmiarach, po jednym dla każdego testu.

Choć mogliśmy wykonać testy z zastosowaniem różnej liczby wątków klienta, 150 wątków jest domyślną liczbą (tak było przynajmniej w czasie pisania książki) zarówno w przypadku serwera Tomcat, jak i *httpd* Apache. Oznacza to, że wielu administratorów użyje domyślnych ustawień głównie z powodu braku czasu na stwierdzenie, jakie jest ich przeznaczenie i jak modyfikować je w standardowy sposób. Ostatecznie dla serwera *httpd* Apache zwiększyliśmy kilka limitów, żeby spróbować znaleźć metodę na poprawienie jego wydajności, gdy klient testujący wysyła jednocześnie maksymalnie 149 żądań. Udało nam się to.

Jest wiele innych rzeczy i metod, które mogliśmy poddać testom. Już samo omówienie innych typowych przypadków użycia wykracza poza zakres książki. Próbujemy jedynie zaprezentować jeden przykład testu porównawczego, który daje trochę przydatnych informacji na temat tego, jak wypada wydajność implementacji serwera WWW Tomcata w porównaniu z osiągami serwera *httpd* Apache w specyficznym i ograniczonym środowisku, a także w przypadku konkretnych testów.

Zewnętrzne dostrajanie

Gdy już wiadomo, jak aplikacja i instancja serwera Tomcat reagują na obciążenie, można zacząć działania mające na celu dostrojenie wydajności. Można wyróżnić dwie podstawowe kategorie dostrajania. Oto one:

Zewnętrzne dostrajanie

Ten typ dostrajania nie uwzględnia żadnych składników innych niż należące do Tomcata. Składnikiem takim nie jest system operacyjny, w którym uruchomiono serwer Tomcat, a także jego wirtualna maszyna Java.

Wewnętrzne dostrajanie

Dostrajanie to dotyczy samego serwera Tomcat, począwszy od zmiany ustawień w plikach konfiguracyjnych, a skończywszy na wprowadzaniu zmian w kodzie źródłowym Tomcata. Do tej kategorii dostrajania należy zaliczyć również modyfikacje aplikacji WWW.

W tym podrozdziale szczegółowo omówimy najbardziej typowe obszary zewnętrznego dostrajania, a w następnym zajmiemy się wewnętrznym dostrajaniem.

Wydajność wirtualnej maszyny Java

Serwer Tomcat nie jest uruchamiany bezpośrednio w systemie operacyjnym komputera. Między jego sprzętem i Tomcatem znajduje się wirtualna maszyna Java i system. W przypadku dowolnej kombinacji systemu operacyjnego i architektury sprzętowej do wyboru jest stosunkowo niewiele kompletnych i pełni zgodnych wirtualnych maszyn Java. W związku z tym większość osób prawdopodobnie pozostanie przy maszynie JVM firmy Sun lub jej implementacji dołączonej do systemu operacyjnego przez jego producenta.

Jeśli celem Czytelnika jest załadowanie najszybszego środowiska uruchomieniowego Java i uzyskanie jak największej wydajności aplikacji WWW, powinno się przetestować Tomcata i aplikację dla każdej wirtualnej maszyny Java dostępnej dla posiadanej kombinacji sprzętu i systemu operacyjnego. Nie należy zakładać, że wirtualna maszyna Java firmy Sun będzie najszybsza, ponieważ często tak nie jest (przynajmniej takie są nasze doświadczenia). Powinno się wypróbować maszyny JVM innych producentów, a nawet różne główne wersje każdej z nich, żeby stwierdzić, w przypadku której z nich konkretna aplikacja WWW działa najszybciej.

Jeśli wybierze się tylko jedną wersję formatu plików klas Java, które muszą być obsługiwane przez używane wirtualne maszyny Java (przykładowo można chcieć skompilować aplikację WWW dla wirtualnych maszyn Java środowiska Java 1.6), można przetestować maszynę JVM każdego dostępnego producenta, która jest zgodna z tym poziomem kodu bajtowego, i wybrać tę, która najlepiej spełnia postawione wymagania. Jeżeli na przykład użyje się środowiska Java 1.6, testom można poddać wirtualne maszyny Java w wersji 1.6 oferowane przez firmy Sun, IBM i BEA. W przypadku jednej z tych maszyn serwer Tomcat i aplikacja WWW będą działały najszybciej. Wszystkie te maszyny JVM są stosowane w środowiskach produkcyjnych przez dużą liczbę użytkowników i każda z nich jest skierowana do trochę innej grupy odbiorców. W dodatku A można znaleźć informacje dotyczące niektórych pakietów JDK, które mogą być dostępne dla używanego systemu operacyjnego.

W ramach ogólnego przykładu poprawiania wydajności w kolejnych podstawowych wersjach wirtualnej maszyny Java jednego producenta można wspomnieć, że dzięki aktualizacji głównej wersji można zyskać 10-procentowy wzrost wydajności. Oznacza to, że po zaktualizowaniu maszyny JVM środowiska Java 1.5 do maszyny JVM środowiska Java 1.6 aplikacja WWW może działać o 10 procent szybciej bez wprowadzania żadnych zmian w jej kodzie. Jest to przypuszczalna wartość, a nie wynik testu porównawczego. Uzyskany wzrost wydajności może się zmieniać zależnie od producenta i wersji testowanej maszyny JVM i funkcji realizowanych przez aplikację WWW.

Choć nowsze wirtualne maszyny Java prawdopodobnie cechują się lepszą wydajnością i mniejszą stabilnością, im dłużej jest dostępna podstawowa wersja maszyny JVM oznaczona jako ostateczna lub stabilna, tym mniej trzeba będzie martwić się o jej niezawodność. Dobrą zasadą

jest uzyskanie najnowszej stabilnej wersji oprogramowania. Wyjątkiem jest sytuacja, gdy najnowsza stabilna wersja jest pierwszą lub drugą w przypadku następnej podstawowej wersji programu. Jeśli na przykład najbardziej aktualna stabilna wersja ma numer 1.7.0, zamiast niej można zdecydować się na wersję 1.6.29, gdy jest bardziej stabilna i działa wystarczająco dobrze.

Często jest tak, że użytkownicy próbują modyfikować parametry startowe wirtualnej maszyny Java Tomcata, żeby szybciej udostępniał strony aplikacji WWW. Choć może to być pomocne, zwykle nie skutkuje znaczącym procentowym wzrostem wydajności. Oto główna przyczyna takiego stanu rzeczy — przed udostępnieniem pakietu JDK producent wirtualnej maszyny Java przeprowadził własne testy, zidentyfikował ustawienia, które zapewniają najlepszą wydajność, a następnie przypisał im rolę domyślnych.

Jeśli zmieni się parametr startowy wirtualnej maszyny Java w celu uaktywnienia ustawienia, które nie jest domyślne, jest ryzyko, że spowoduje się spowolnienie maszyny JVM. Czytelnik został ostrzeżony! Jeśli jednak chciałoby się sprawdzić, jakie ustawienia wirtualnej maszyny Java firmy Sun można zmodyfikować, należy zajrzeć na stronę internetową dostępną pod adresem *http://www.md.pp.ru/~eu/jdk6options.html*.

Wyjątkiem jest przydzielanie pamięci sterty wirtualnej maszynie Java. Domyślnie producenci decydują się, żeby na początku maszynie JVM przydzielić niewielką ilość pamięci (w przypadku maszyny JVM firmy Sun są to 32 MB). Jeśli aplikacja Java wymaga większej ilości pamięci, po wstrzymaniu jej pracy rozmiar pamięci sterty maszyny JVM jest odpowiednio zwiększany. Wirtualna maszyna Java może kilkakrotnie niewielkimi skokami zwiększać ilość wykorzystywanej pamięci aż do momentu osiągnięcia ustalonego maksymalnego rozmiaru pamięci sterty. Jeżeli dojdzie do tego, gdy Tomcat udostępnia strony aplikacji WWW, czas oczekiwania na odpowiedzi znacznie się wydłuży w przypadku każdego klienta WWW, którego żądania nie zostały zrealizowane w momencie rozpoczęcia przerwy w oferowaniu zasobów przez serwer Tomcat. Aby uniknąć takich przerw, jako minimalny i maksymalny rozmiar sterty można ustawić taką samą wartość. Dzięki temu w czasie pracy maszyna JVM nie będzie próbowała zwiększać wielkości pamięci sterty. Aby tak było, w przypadku wirtualnej maszyny Java Tomcata dla zmiennej środowiskowej `JAVA_OPTS` wystarczy ustawić wartości `-Xms512M -Xmx512M` (oznacza to, że minimalny i maksymalny rozmiar sterty powinien wynieść `512 MB`). Zależnie od ilości pamięci dostępnej po załadowaniu systemu należy określić odpowiednią wartość dla tej zmiennej.

Można też spróbować przetestować różne ustawienia algorytmu zbierającego zwolnione obszary pamięci. Jak już wcześniej wspomnieliśmy, może się okazać, że domyślne ustawienia zawsze są najszybsze. Jednak do momentu wykonania testów nigdy nie uzyska się całkowitej pewności. W dokumentacji testowanej wirtualnej maszyny Java należy poszukać parametru startowego, który uaktywni inny algorytm zbierający zwolnione obszary pamięci, ponieważ ustawienie to jest specyficzne dla implementacji maszyny JVM. Aby wirtualna maszyna Java Tomcata była w żądany sposób ładowania, parametr należy uwzględnić w zmiennej środowiskowej `JAVA_OPTS`.

Wydajność systemu operacyjnego

A co z systemem operacyjnym? Czy system serwera jest zoptymalizowany pod kątem uruchamiania serwera WWW o dużym obciążeniu? Oczywiście, w czasie prac projektowych przed różnymi systemami operacyjnymi postawiono bardzo odmienne cele. Przykładowo w przypadku

systemu OpenBSD skoncentrowano się na bezpieczeństwie. Z tego powodu w jądrze dla wielu limitów ustawiono niskie wartości, żeby zapobiec różnego rodzaju atakom typu DoF (*Denial-of--Service*). Jedno z mott towarzyszących systemowi OpenBSD brzmi: „domyślnie bezpieczny". Limity te najprawdopodobniej będą musiały być zwiększone w celu umożliwienia efektywnej pracy mocno obciążonemu serwerowi WWW.

Z kolei system Linux ma być prosty w obsłudze. W związku z tym w jego przypadku są ustawione wyższe limity. Domyślnie po zainstalowaniu systemu BSD jego jądro jest wersji podstawowej. Oznacza to, że większość sterowników do jądra jest statycznie dołączana. Poza tym, że na początku stanowi to ułatwienie, jeśli tworzy się niestandardowe jądro w celu zwiększenia określonych limitów, można również usunąć z niego niepotrzebne urządzenia. W przypadku jądra systemu Linux większość sterowników jest dynamicznie ładowana. Z kolei ze względu na to, że sama pamięć tanieje, mniej istotny stał się argument przemawiający za stosowaniem ładowalnych sterowników urządzeń. Ważne jest, żeby dysponować mnóstwem pamięci i w jak największej ilości udostępniać ją serwerowi.

Choć obecnie pamięć nie jest droga, nie należy kupować tanich układów. Markowa pamięć kosztuje tylko trochę więcej i różnicę w cenie odpłaci w postaci niezawodności.

Jeżeli korzysta się z dowolnej odmiany systemu Microsoft Windows, trzeba postarać się o jego wersję serwerową (czyli na przykład zamiast wersji Pro systemu Windows Vista zastosować wersję Server). W przypadku wersji systemu Windows innych niż serwerowa warunki umowy licencyjnej i (lub) kod samego systemu operacyjnego mogą ograniczać liczbę dostępnych użytkowników lub połączeń sieciowych bądź nakładać inne restrykcje dotyczące tego, co można uruchomić. Ponadto z oczywistych względów bezpieczeństwa trzeba zadbać o częste pobieranie z witryny Microsoftu najnowszych pakietów Service Pack (dotyczy to każdego systemu, lecz szczególne istotne jest w przypadku systemu Windows).

Wewnętrzne dostrajanie

W tym podrozdziale dokładnie przedstawiono specyficzny zestaw metod, które przyczynią się do tego, że instancja serwera Tomcat będzie działać szybciej, niezależnie od używanego systemu operacyjnego lub wirtualnej maszyny Java. W wielu przypadkach nie można kontrolować systemu lub maszyny JVM załadowanej na komputerze, na którym wdraża się oprogramowanie. W takich sytuacjach powinno się podać zalecenia uwzględniające to, co szczegółowo omówiono w poprzednim podrozdziale. Niezależnie od tego powinno być możliwe wprowadzanie zmian w samym serwerze Tomcat. Poniżej zaprezentowaliśmy rzeczy, które według nas najlepiej nadają się do rozpoczęcia procesu wewnętrznego dostrajania Tomcata.

Wyłączenie funkcji sprawdzania adresów DNS

Gdy aplikacja WWW chce zarejestrować informacje o kliencie, może zapisać jego numeryczny adres IP lub w bazie danych serwera DNS (*Domain Name Service*) poszukać rzeczywistej nazwy hosta. Operacje sprawdzania adresów DNS generują ruch sieciowy, do czego przyczynia się czekanie na odpowiedzi zwrotne z wielu serwerów, które mogą być znacznie oddalone i niedostępne. W efekcie występują opóźnienia. W celu wyeliminowania ich można wyłączyć funkcję

sprawdzania adresów DNS. Jeśli tak się postąpi, każdorazowo gdy aplikacja WWW wywoła metodę `getRemoteHost()` obiektu żądania HTTP, uzyska jedynie adres IP. Coś takiego konfiguruje się w obrębie elementu `Connector` aplikacji, zlokalizowanego w pliku *server.xml* serwera Tomcat. W przypadku typowego złącza `java.io` HTTP 1.1 należy skorzystać z atrybutu `enableLookups`. W pliku *server.xml* wystarczy znaleźć następujące wiersze:

```
<!-- Definiowanie złącza HTTP/1.1 bez SSL dla portu 8080 -->
    <Connector port="8080" maxHttpHeaderSize="8192"
               maxThreads="150" minSpareThreads="25" maxSpareThreads="75"
enableLookups="true" redirectPort="8443" acceptCount="100"
               connectionTimeout="20000" disableUploadTimeout="true" />
```

Trzeba jedynie wartość argumentu `enableLookups` zmienić z `true` na `false` i zrestartować Tomcata. Od tego momentu nie będą już sprawdzane adresy DNS i nie wystąpią związane z tym opóźnienia!

Jeśli nie jest wymagana w pełni kwalifikowana nazwa hosta każdego klienta HTTP, który łączy się z witryną, zalecamy wyłączenie w środowisku produkcyjnym funkcji sprawdzania adresów DNS. Trzeba pamiętać, że później zawsze można poszukać nazw hostów poza obrębem Tomcata. Przez zrezygnowanie ze sprawdzania adresów DNS nie tylko oszczędza się przepustowość sieci, skraca czas wyszukiwania i redukuje zużycie pamięci, ale też w przypadku witryn, w obrębie których spory ruch sieciowy generuje znaczną ilość rejestrowanych danych, można zyskać zauważalną pojemność przestrzeni dyskowej. Choć w przypadku witryn o niewielkim ruchu sieciowym wyłączenie funkcji sprawdzania adresów DNS może nie mieć tak dużego wpływu, też jest to dobry pomysł. Jak często w ciągu nocy witryny o znikomym ruchu sieciowym stały się witrynami bardzo obciążonymi?

Dostosowywanie liczby wątków

Następny parametr wydajnościowy znajdujący się wewnątrz elementu `Connector` aplikacji WWW pozwala określić używaną przez nią liczbę wątków obsługujących żądania. Domyślnie serwer Tomcat używa puli wątków w celu zapewnienia krótkiego czasu odpowiedzi na przychodzące żądania. Podobnie do innych języków programowania w przypadku języka Java wątek jest niezależnym obiektem sterującym przepływem danych prowadzącym interakcje z systemem operacyjnym i własną pamięcią lokalną, a także z pamięcią współużytkowaną przez wszystkie wątki procesu. Dzięki temu projektanci mogą zapewnić szczegółową organizację kodu, który będzie dobrze reagował na wiele przychodzących żądań.

Możliwe jest kontrolowanie liczby przydzielanych wątków przez zmianę wartości argumentów `minThreads` i `maxThreads` elementu `Connector`. Choć domyślne wartości tych argumentów są wystarczające w przypadku typowych instalacji, może być konieczne zwiększenie ich, gdy witryna zacznie się rozrastać. Wartość argumentu `minThreads` powinna być wystarczająco duża, żeby było obsługiwane minimalne obciążenie. Oznacza to, że jeśli w porze dnia o niewielkim natężeniu ruchu w ciągu sekundy otrzymuje się pięć żądań i przetworzenie każdego z nich zajmuje sekundę, wymagane będzie przydzielenie jedynie pięciu wątków. Gdy później, w ciągu dnia, wzrośnie obciążenie witryny, konieczne będzie przypisanie większej liczby wątków (aż do maksymalnej liczby określonej przez wartość atrybutu `maxThreads`). Aby zapobiec skokom obciążenia (lub atakom typu DoF przeprowadzanym przez użytkownika z niecnymi zamiarami) spowodowanym przez „bombardowanie" serwera mające na celu wymuszenie przekroczenia limitu maksymalnej ilości pamięci przydzielonej wirtualnej maszynie Java, trzeba ustalić górny limit liczby wątków.

Najlepszą metodą określenia dla tych atrybutów optymalnych wartości jest wypróbowanie dla każdego z atrybutów wielu różnych ustawień i przetestowanie ich przy symulowanym obciążeniu sieciowym wraz z obserwacją czasów odpowiedzi i wykorzystania pamięci. Każda kombinacja sprzętu, systemu operacyjnego i maszyny JVM może zachowywać się inaczej, a ponadto różne jest natężenie ruchu w przypadku poszczególnych witryn internetowych. W związku z tym nie istnieje jednoznaczna metoda pozwalająca określić minimalną i maksymalną liczbę wątków.

Przyspieszanie stron JSP

Gdy strona JSP jest po raz pierwszy używana, jest konwertowana na kod źródłowy serwletu Java, który następnie musi zostać skompilowany do postaci kodu bajtowego Java.

Innym rozwiązaniem jest zupełne zrezygnowanie ze stron JSP i wykorzystanie niektórych mechanizmów szablonów Java, które obecnie są dostępne. Choć oczywiście jest to decyzja na dużą skalę, wiele osób przekonało się, że przynajmniej warto było przeanalizować taki wariant. W celu uzyskania dokładnych informacji na temat innych języków szablonów, których można użyć w przypadku Tomcata, należy sięgnąć do książki *Java Servlet Programming* autorstwa Jasona Huntera i Williama Crawforda (wydawnictwo O'Reilly).

Prekompilacja stron JSP przez zażądanie ich

Ponieważ standardowo strona JSP jest kompilowana, gdy zostanie po raz pierwszy użyta za pośrednictwem witryny WWW, zamiast czekać, aż pierwszy użytkownik odwiedzi stronę JSP po jej zainstalowaniu i uaktualnieniu, można zdecydować się na prekompilację tej strony. W ten sposób łatwiej będzie można zapewnić, że nowa strona JSP zadziała w obrębie serwera produkcyjnego tak jak w przypadku komputera testowego.

W katalogu *bin/* serwera Tomcat znajduje się skrypt o nazwie *jspc*, który wygląda na taki, który mógłby zostać użyty do prekompilacji stron JSP. Jednak tak nie jest. Skrypt ten dokonuje translacji z kodu źródłowego JSP do kodu Java, lecz nie kompiluje go. Ponadto skrypt generuje wynikowy plik z kodem źródłowym Java w bieżącym katalogu, a nie w katalogu roboczym aplikacji WWW. Skrypt przede wszystkim przyda się osobom zajmującym się debugowaniem stron JSP.

Najprostszą metodą zapewnienia prekompilacji dowolnej strony JSP jest po prostu uzyskanie dostępu do niej za pośrednictwem klienta WWW. W ten sposób zagwarantuje się, że plik zostanie zamieniony na serwlet, skompilowany, a następnie uruchomiony. Rozwiązanie to jest też korzystne, ponieważ pozwala dokładnie symulować to, jak użytkownik uzyskuje dostęp do strony JSP. Dzięki temu można stwierdzić, jakie użytkownik wykona operacje. Oczywiście, takie działania etapu projektowania najlepiej przeprowadzać w środowisku do tego przeznaczonym, a nie na serwerze produkcyjnym.

Prekompilacja stron JSP podczas ładowania aplikacji WWW

Inną znakomitą, lecz rzadko stosowaną cechą specyfikacji Java Servlet Specification, jest to, że nakazuje ona kontenerom serwletów umożliwienie aplikacjom WWW określenia stron JSP, które powinny być prekompilowane w czasie uruchamiania aplikacji.

Jeśli na przykład oczekuje się, że plik *index.jsp* (zlokalizowany w głównym katalogu aplikacji WWW) zawsze będzie prekompilowany w czasie ładowania aplikacji, dla tego pliku w pliku *web.xml* można wstawić znacznik `<servlet>`.

```
<web-app xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/
javaee/web-app_2_5.xsd"
    version="2.5">

<servlet>
    <servlet-name>index.jsp</servlet-name>
    <jsp-file>/index.jsp</jsp-file>
    <load-on-startup>0</load-on-startup>
  </servlet>

</web-app>
```

W efekcie Tomcat automatycznie dokona prekompilacji pliku *index.jsp* podczas uruchamiania aplikacji WWW. Pierwsze żądanie dotyczące tego pliku zostanie przemapowane na plik klasy serwletu prekompilowany dla strony JSP.

Skonfigurowanie w ten sposób prekompilacji dla aplikacji WWW spowoduje, że wszystkie operacje kompilowania stron JSP będą wykonywane na etapie uaktywniania aplikacji WWW, niezależnie od tego, czy strony będą zażądane przez klientów WWW, czy nie. Każda strona JSP zdeklarowana w ten sposób w pliku *web.xml* będzie prekompilowana. Wadą takiego rozwiązania jest to, że czas ładowania aplikacji WWW zawsze wydłuży się, ponieważ każda określona strona musi być prekompilowana, zanim aplikacja zostanie udostępniona klientom WWW.

Poza tym znacznik kontenerowy `<load-on-startup>` powinien zawierać dodatnią liczbę całkowitą. Jest to ogólna metoda ustalania kolejności prekompilacji. Im niższą dla strony JSP ustawi się tę wartość, tym wcześniej będzie prekompilowana podczas uruchamiania aplikacji WWW.

Gdy w ten sposób dokona się prekompilacji stron JSP, mogą być szybciej dostępne dla pierwszego klienta WWW, który po pierwszym lub kolejnym wdrożeniu aplikacji WWW zażąda tych stron. Jednak strony JSP kompilowane na etapie procesu budowania (przed wdrożeniem) aplikacji będą trochę szybciej dostępne w przypadku wszystkich żądań, nawet tych, które zostały wygenerowane jako kolejne po pierwszych żądaniach dotyczących poszczególnych stron JSP.

Prekompilacja stron JSP na etapie procesu budowania za pomocą narzędzia JspC

Oto kilka uzasadnionych (w czasie, gdy pisano książkę) powodów dokonywania prekompilacji stron JSP na etapie procesu budowania aplikacji WWW:

- Trzeba w pełni wykorzystać możliwości aplikacji WWW. Strony JSP kompilowane podczas procesu budowania aplikacji WWW działają szybciej od stron skompilowanych w obrębie serwera Tomcat już po wdrożeniu aplikacji. Przede wszystkim kod bajtowy klas Java wygenerowany w obu wariantach naprawdę powinien być identyczny. Jeśli tak nie jest, różnica będzie znikoma (z pewnością nie warta wprowadzania większych zmian na etapie wdrażania, takich, jakie są wymagane w celu prekompilacji stron JSP przed wdrożeniem aplikacji). Czas, jaki zajmuje serwerowi Tomcat skompilowanie oryginalnej strony JSP, zwykle jest krótki. Poza tym operacja taka ma miejsce tylko w przypadku pierwszego żądania dotyczącego poszczególnych stron JSP, które wygenerowano po pierwszym lub kolejnym wdrożeniu aplikacji WWW. Wszystkie pozostałe żądania udostępnienia strony

JSP są obsługiwane przez skompilowaną i załadowaną klasę serwletu JSP (strony JSP są kompilowane do postaci serwletów Java). Jednak ze względu na to, że strony JSP skompilowane przed wdrożeniem aplikacji WWW są mapowane na przestrzeń URI zdefiniowaną w pliku *web.xml*, serwer Tomcat jest w stanie kierować żądania do stron trochę szybciej, niż gdyby strona JSP została skompilowana w czasie pracy aplikacji. Wynika to stąd, że w przypadku kompilowania stron JSP na etapie uruchamiania aplikacji WWW utworzone serwlety muszą być najpierw mapowane na przestrzeń URI za pomocą standardowego modułu mapującego URI, który kieruje żądanie do klasy `JspServlet` serwera Tomcat, a ta następnie mapuje żądanie do wymaganej przez nie strony JSP. Warto zauważyć, że strony JSP kompilowane w czasie działania aplikacji są mapowane przy wykorzystaniu dwóch warstw pośredniczących (dwa odrębne moduły mapujące). Z kolei prekompilowane strony JSP są mapowane tylko za pośrednictwem pierwszej takiej warstwy. W celu stwierdzenia różnicy w wydajności należy określić wydajność w przypadku dwóch odmiennych wariantów mapowania na przestrzeń URI. Okazuje się, że prekompilowane strony JSP zwykle działają szybciej o około 4 procent. Prekompilowanie stron JSP przed wdrożeniem aplikacji WWW pozwoli zaoszczędzić niewielką ilość czasu potrzebnego na skompilowanie każdej strony JSP aplikacji wymaganej przez pierwsze żądanie, a także uzyskać 4-procentowy wzrost wydajności w przypadku każdego kolejnego żądania dotyczącego strony JSP. Wchodzący w skład serwera Tomcat 4.1.x moduł mapujący żądania stron JSP w czasie działania aplikacji WWW był zauważalnie wolniejszy od modułu mapującego serwlety zdefiniowane w pliku *web.xml*. W tym przypadku warto było dokonać prekompilacji stron JSP przed wdrożeniem aplikacji WWW. Dzięki temu w naszych testach strony JSP okazały się szybsze w przybliżeniu o 12 procent. Jednak w przypadku serwera Tomcat w wersji 5.0.x lub nowszej ten wzrost wydajności został zmniejszony do około 4 procent lub mniej.

- Dzięki prekompilacji stron JSP na etapie budowania aplikacji WWW lub tworzenia jej pakietu dla stron składnia jest sprawdzana podczas procesu ich kompilowania. Oznacza to, że można być pewnym, że przed wdrożeniem aplikacji WWW strony JSP zostaną przynajmniej skompilowane bez żadnych błędów składni. Jest to znakomity sposób na uniknięcie sytuacji, w której aplikację WWW wdrożono na serwerze produkcyjnym tylko po to, żeby później stwierdzić (po użytkowniku, który jako pierwszy zażądał strony), że jedna ze stron JSP zawiera błąd składni. Ponadto przez szukanie błędów na etapie tworzenia kodu projektant ma możliwość znacznie szybszego zidentyfikowania ich i usunięcia. W ten sposób skraca się czas trwania prac projektowych. Postępując tak, nie zapobiegnie się każdemu rodzajowi błędu, ponieważ skompilowana strona JSP może zawierać błędy logiczne ujawniające się w czasie działania aplikacji. Jednak przynajmniej w środowisku projektowym można wychwycić wszystkie błędy składni.
- Jeśli w aplikacji WWW znajduje się bardzo dużo plików JSP, z których każdy jest raczej pokaźnych rozmiarów (mamy nadzieję, że Czytelnik nie kopiuje i nie wkleja dużej ilości danych z jednej strony JSP do wielu innych stron; zamiast tego powinien skorzystać z funkcji dołączania zawartości), sumaryczny czas początkowej kompilacji wszystkich stron może być znaczący. Jeżeli tak jest, w przypadku serwera produkcyjnego można zyskać na czasie przez prekompilację stron JSP przed wdrożeniem aplikacji WWW. Jest to szczególnie pomocne w przypadku dużego natężenia ruchu sieciowego. Jeśli nie skorzysta się z prekompilacji, czasy odpowiedzi serwera dość znacznie wydłużą się, gdy serwer przeprowadzi początkową kompilację wielu stron JSP w czasie uruchamiania aplikacji WWW po raz pierwszy.

- Jeżeli serwer dysponuje niewielką ilością zasobów (na przykład, gdy wirtualnej maszynie Java przydzielono mało pamięci RAM lub serwer nie oferuje Tomcatowi zbyt wielu cykli procesorowych), można chcieć zupełnie zrezygnować z kompilowania stron JSP na serwerze. W tym przypadku kompilację można wykonać w środowisku projektowym i wdrożyć wyłącznie skompilowane serwlety. Dzięki temu po wdrożeniu każdej nowej kopii aplikacji WWW zmniejszy się poziom wykorzystania pamięci i procesora przez pierwsze żądanie dotyczące pliku poszczególnych stron JSP.
- Projektuje się aplikację WWW ze stronami JSP, która zostanie sprzedana klientom niewymagającym kodu źródłowego JSP. Jeśli możliwe by było przekazanie klientom aplikacji zawierającej tylko skompilowane serwlety, można by utworzyć ją przy wykorzystaniu oryginalnych stron JSP, a następnie dostarczyć odbiorcom wraz ze skompilowanymi serwletami stron JSP. W tym przypadku prekompilacja przed przekazaniem aplikacji klientowi pełni rolę mechanizmu maskowania kodu źródłowego. Trzeba jednak mieć świadomość tego, że skompilowane pliki klas Java dość łatwo poddać dekompilacji do postaci czytelnego kodu źródłowego Java. W czasie, gdy pisano książkę, nie istniała metoda pozwalająca całkowicie zdekompilować plików klas do postaci kodu źródłowego stron JSP.
- Począwszy od wersji 5.5 serwera Tomcat, do obsługi stron JSP skompilowanych w czasie działania aplikacji WWW nie jest już potrzebny zestaw JDK z wbudowanym kompilatorem kodu źródłowego Java. Tomcat w wersji 5.5 i nowszej dysponuje kompilatorem Eclipse JDT, który jest kompilatorem kodu Java napisanym w czystym języku Java. Ponieważ kompilator JDT dołączono do serwera Tomcat, może on zawsze skompilować strony JSP do postaci serwletów, nawet wtedy, gdy serwer uruchomiono w środowisku JRE, a nie JDK.

Przykład 4.4 prezentuje zawartość pliku narzędzia Ant realizującego proces budowania. Pliku można użyć do skompilowania plików stron JSP aplikacji WWW na etapie jej budowania.

Przykład 4.4. Plik procesu budowania narzędzia Ant o nazwie precompile-jsps.xml

```
<project name="pre-compile-jsps" default="compile-jsp-servlets">

  <!-- Prywatne właściwości. -->
  <property name="webapp.dir" value="${basedir}/webapp-dir"/>
  <property name="tomcat.home" value="/opt/tomcat"/>
  <property name="jspc.pkg.prefix" value="com.mojafirma"/>
  <property name="jspc.dir.prefix" value="com/mojafirma "/>

<!-- Właściwości kompilacji. -->
  <property name="debug" value="on"/>
  <property name="debuglevel" value="lines,vars,source"/>
  <property name="deprecation" value="on"/>
  <property name="encoding" value="ISO-8859-1"/>
  <property name="optimize" value="off"/>
  <property name="build.compiler" value="modern"/>
  <property name="source.version" value="1.5"/>

  <!-- Ścieżki inicjalizacyjne. -->
  <path id="jspc.classpath">
    <fileset dir="${tomcat.home}/bin">
      <include name="*.jar"/>
    </fileset>
    <fileset dir="${tomcat.home}/server/lib">
      <include name="*.jar"/>
    </fileset>
    <fileset dir="${tomcat.home}/common/i18n">
      <include name="*.jar"/>
```

```
    </fileset>
    <fileset dir="${tomcat.home}/common/lib">
      <include name="*.jar"/>
    </fileset>
    <fileset dir="${webapp.dir}/WEB-INF">
      <include name="lib/*.jar"/>
    </fileset>
    <pathelement location="${webapp.dir}/WEB-INF/classes"/>
    <pathelement location="${ant.home}/lib/ant.jar"/>
    <pathelement location="${java.home}/../lib/tools.jar"/>
  </path>
  <property name="jspc.classpath" refid="jspc.classpath"/>

  <!-- ======================================================================
-->
  <!-- Generowanie z plików stron JSP kodu źródłowego Java i pliku web.xml.    -->
  <!-- ======================================================================
-->
  <target name="generate-jsp-java-src">
    <mkdir dir="${webapp.dir}/WEB-INF/jspc-src/${jspc.dir.prefix}"/>
    <taskdef classname="org.apache.jasper.JspC" name="jasper2">
      <classpath>
        <path refid="jspc.classpath"/>
      </classpath>
    </taskdef>
    <touch file="${webapp.dir}/WEB-INF/jspc-web.xml"/>
    <jasper2 uriroot="${webapp.dir}"
             package="${jspc.pkg.prefix}"
             webXmlFragment="${webapp.dir}/WEB-INF/jspc-web.xml"
             outputDir="${webapp.dir}/WEB-INF/jspc-src/${jspc.dir.prefix}"
             verbose="1"/>
  </target>

<!-- ============================================================= -->
  <!-- Kompilowanie (tworzenie plików klas Java) kodu źródłowego                -->
  <!-- serwletu JSP wygenerowanego przez zadanie JspC.                          -->
  <!-- ============================================================= -->
  <target name="compile-jsp-servlets" depends="generate-jsp-java-src">
    <mkdir dir="${webapp.dir}/WEB-INF/classes"/>
    <javac srcdir="${webapp.dir}/WEB-INF/jspc-src"
           destdir="${webapp.dir}/WEB-INF/classes"
           includes="**/*.java"
           debug="${debug}"
           debuglevel="${debuglevel}"
           deprecation="${deprecation}"
           encoding="${encoding}"
           optimize="${optimize}"
           source="${source.version}">
      <classpath>
        <path refid="jspc.classpath"/>
      </classpath>
    </javac>
  </target>

  <!-- ============================================================= -->
  <!-- Usuwanie kodu źródłowego prekompilowanych stron JSP,                     -->
  <!-- a także klas i pliku jspc-web.xml.                                       -->
  <!-- ============================================================= -->
  <target name="clean">
    <delete dir="${webapp.dir}/WEB-INF/jspc-src"/>
    <delete dir="${webapp.dir}/WEB-INF/classes/${jspc.dir.prefix}"/>
    <delete file="${webapp.dir}/WEB-INF/jspc-web.xml"/>
  </target>

</project>
```

Jeśli powyższy kod XML umieści się w pliku o nazwie, takiej jak *precompile-jsps.xml*, będzie można go testować razem z dowolnym już istniejącym plikiem *build.xml*. W razie potrzeby ten kod XML można dołączyć do pliku *build.xml*.

Plik procesu tworzenia odszuka wszystkie pliki stron JSP aplikacji WWW, skompiluje je do postaci klas serwletów i utworzy dla nich mapowania. Wygenerowane mapowania serwletów muszą trafić do pliku *WEB-INF/web.xml* aplikacji WWW. Jednak trudne byłoby utworzenie pliku procesu budowania narzędzia Ant, który każdorazowo po uruchomieniu mógłby w powtarzalny sposób umieszczać mapowania serwletów w pliku *web.xml*. Zamiast tego użyliśmy dołączenia encji XML, dzięki czemu generowane mapowania serwletów trafią do nowego pliku każdorazowo po wywołaniu pliku procesu budowania. Ponadto utworzony plik mapowań może być dołączony do pliku *web.xml* za pośrednictwem mechanizmu bazującego na dołączeniu encji XML. Aby skorzystać z tego mechanizmu, na samym początku pliku *WEB-INF/web.xml* aplikacji WWW musi zostać wstawiona specjalna deklaracja encji, a także odwołanie do niej w miejscu pliku, w którym ma być dołączona zawartość pliku mapowań serwletów. Poniżej pokazano, jak wygląda plik *web.xml* pustej serwletowej aplikacji WWW w wersji 2.5 po wprowadzeniu takich modyfikacji.

```
<!DOCTYPE jspc-webxml [
  <!ENTITY jspc-webxml SYSTEM "jspc-web.xml">
]>

<web-app xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/
javaee/web-app_2_5.xsd"
    version="2.5">

<!-- W tym miejscu dołączamy mapowania generowane przez zadanie JspC. -->
  &jspc-webxml;

  <!-- Tu trafia niewygenerowana zawartość pliku web.xml. -->

</web-app>
```

Trzeba się upewnić, czy plik *web.xml* aplikacji WWW na samym początku ma wbudowany dokument DTD (znacznik `DOCTYPE`), a poniżej deklarację schematu `web-app` serwletów w wersji 2.5. Gdy następnie będzie się zamierzało wstawić do pliku *web.xml* wygenerowane mapowania serwletów, należy umieścić w nim odwołanie encji `&jspc-webxml;`. Trzeba pamiętać, że odwołanie takie zaczyna się od znaku `&`, za którym jest nazwa encji i średnik.

W celu zastosowania pliku procesu budowania wystarczy poddać go edycji i dla wszystkich właściwości zlokalizowanych na samym początku pliku ustawić wartości właściwe dla konkretnej konfiguracji, a następnie uruchomić plik za pomocą poniższego polecenia.

```
$ ant -f pre-compile-jsps.xml
Buildfile: pre-compile-jsps.xml

generate-jsp-java-src:
  [jasper2] Sep 27, 2008 10:47:15 PM org.apache.jasper.xmlparser.MyEntityResolver
resolveEntity
  [jasper2] SEVERE: Invalid PUBLIC ID: null
  [jasper2] Sep 27, 2007 10:47:17 PM org.apache.jasper.JspC processFile
  [jasper2] INFO: Built File: /index.jsp

compile-jsp-servlets:
    [javac] Compiling 1 source file to /home/jasonb/myproject/webapp-dir/WEB-INF/
```

```
classes

BUILD SUCCESSFUL
Total time: 7 seconds
```

Wszystkie pliki stron JSP znajdujące się w katalogu aplikacji WWW zostaną skompilowane do postaci serwletów. Gdy wdroży się aplikację, żądania dotyczące stron JSP będą mapowane na skompilowane serwlety. Jeśli pojawi się komunikat `SEVERE: Invalid PUBLIC ID: null`, należy go zignorować, ponieważ jest fałszywy. Jeżeli planuje się usunąć skompilowane serwlety, a także ich wygenerowany kod źródłowy Java i mapowania, wystarczy wykonać następujące polecenie:

```
$ ant -f pre-compile-jsps.xml clean
```

Ten plik procesu budowania nie wykonuje jednej operacji, czyli usuwania wszystkich plików stron JSP aplikacji WWW po ich skompilowaniu. Ponieważ nie chcieliśmy przypadkowo usunąć plików stron JSP, świadomie je pozostawiliśmy. Plik procesu budowania utworzony przez Czytelnika powinien usunąć pliki stron JSP przed wdrożeniem aplikacji WWW. Jeśli się o tym zapomni i omyłkowo pozostawi pliki stron JSP w obrębie wdrożonej aplikacji, żaden z nich nie powinien być udostępniony przez serwer Tomcat, gdyż plik *web.xml* wyraźnie instruuje go, żeby stosował skompilowane klasy serwletów.

Planowanie obciążenia

Planowanie obciążenia jest następną istotną częścią dostrajania wydajności produkcyjnego serwera Tomcat. Skala poczynionych działań mających na celu dostrajanie i testowanie plików konfiguracyjnych w rzeczywistości nie będzie pomocna, jeżeli nie będzie się dysponować sprzętem i przepustowością wymaganą przez witrynę do obsługi spodziewanego natężenia ruchu sieciowego.

Oto ogólna definicja planowania obciążenia pasująca do kontekstu niniejszego podrozdziału: *planowanie obciążenia* jest szacowaniem wymagań witryny dotyczących sprzętu komputerowego, systemu operacyjnego i przepustowości. Sprowadza się to do analizowania i (lub) przewidywania całkowitego natężenia ruchu sieciowego, który będzie musiał być obsłużony przez witrynę, podejmowania decyzji związanych z dopuszczalnymi charakterystykami usług, a także szukania odpowiedniego sprzętu i systemów operacyjnych spełniających lub przekraczających wymagania określone przez usługi w stosunku do oprogramowania serwera. W tym przypadku oprogramowaniem takim jest serwer Tomcat, jak również wszelkie zewnętrzne serwery WWW i narzędzia równoważące obciążenie, które umiejscowiono „przed" Tomcatem.

Jeśli przed nabyciem i wdrożeniem serwerów produkcyjnych nie przeprowadzi się planowania obciążenia, nie będzie wiadomo, czy sprzęt serwerowy poradzi sobie z ruchem sieciowym obciążającym witrynę internetową. Co gorsza, nie będzie się świadomym problemu do momentu zamówienia i zapłacenia za aplikacje, a następnie wdrożenia ich na zakupionym sprzęcie. Zwykle będzie za późno na zbyt radykalne zmiany kierunku działań. Choć zazwyczaj można dodać pojemniejszy dysk twardy, a nawet zamówić więcej serwerów, czasami ogólne koszty będą mniejsze, gdy od razu na początku kupi się i (lub) będzie utrzymywać mniej komputerów pełniących rolę serwerów.

Im wyższe natężenie ruchu sieciowego lub większe obciążenie generowane dla jednego żądania klienta, tym ważniejsze staje się planowanie obciążenia. Do niektórych witryn jest kierowany tak duży ruch sieciowy, że poradzić sobie z nim w całości przy zapewnieniu rozsądnych

limitów czasu odpowiedzi może tylko klaster złożony z serwerów. Z kolei w przypadku mniej obciążonych witryn mniejszym problemem będzie znalezienie sprzętu spełniającego postawione wymagania. Choć prawdą jest, że po wystąpieniu problemu dodanie większej ilości sprzętu lub bardziej wydajnych urządzeń zwykle pozwoli go usunąć, szczególnie w przypadku dużego ruchu sieciowego może to być rozwiązanie, którego koszt będzie nie do przyjęcia. W przypadku większości firm jest tak, że im niższe są koszty sprzętu (włącznie z kosztem jego bieżącej konserwacji, mającej miejsce od momentu zakupu), tym większe mogą być zyski. Innym godnym uwagi wskaźnikiem jest wydajność pracowników. Jeśli na przykład wydajniejszy sprzęt przyczyni się do tego, że projektanci wykonają swoją pracę o 20 procent szybciej, zależnie od wielkości zespołu, nabycie od razu wydajniejszych lub szybszych urządzeń może być warte różnicy w ich cenie.

Planowanie obciążenia zwykle ma również miejsce przy modernizacjach. Przed zamówieniem sprzętu, który ma zastąpić istniejące krytyczne serwery, prawdopodobnie dobrym pomysłem będzie zebranie informacji na temat tego, co firma potrzebuje, w oparciu o kryteria stawiane modernizacji, typowe obciążenie generowane przez ruch sieciowy, wymagania oprogramowania itp.

Można wyróżnić co najmniej kilka typowych metod podejmowania decyzji podczas planowania obciążenia. W praktyce spotkaliśmy się z dwoma podstawowymi metodami — anegdotyczną i akademicką (na przykład planowanie obciążenia w przedsiębiorstwie).

Anegdotyczne planowanie obciążenia

Anegdotyczne planowanie obciążenia jest w pewnym sensie uproszczoną odmianą planowania obciążenia, które z założenia nie ma być ścisłe, lecz wystarczająco dokładne, żeby pozwoliło firmie uniknąć sytuacji będących następstwem zupełnego zrezygnowania z planowania obciążenia. Metoda ta bazuje na tendencjach dotyczących obciążenia i wydajności, które są wynikiem dotychczasowego doświadczenia zdobytego przez przedstawicieli branżowych. Przykładowo na podstawie posiadanej wiedzy (mamy nadzieję, że bazującej na jakiejś innej, rzeczywistej witrynie internetowej) można jak najdokładniej przyjąć, jaki w chwilach szczytowego obciążenia będzie ruch sieciowy wychodzący z witryny, a następnie założoną wartość podwoić. Wartość ta jest dla witryny nową wymaganą przepustowością, która musi zostać obsłużona przez zakupiony i wdrożony sprzęt. Większość osób będzie w ten sposób planować obciążenie, ponieważ jest to szybka metoda, która wymaga niewiele czasu i nakładu pracy.

Planowanie obciążenia w przedsiębiorstwie

Planowanie obciążenia w przedsiębiorstwie ma być bardziej precyzyjne i zajmować więcej czasu. Metoda ta jest niezbędna w przypadku witryn o bardzo dużym natężeniu ruchu sieciowego, często połączonym z dużym obciążeniem przypadającym na jedno żądanie. Takie szczegółowe planowanie obciążenia jest konieczne w celu zapewnienia jak najniższego kosztu sprzętu i przepustowości sieci przy jednoczesnym zagwarantowaniu jakości usługi oferowanej przez firmę lub spełnienia przez nią warunków określonych w kontrakcie. Oprócz wielokrotnych testów i modelowania zwykle wiąże się to z zastosowaniem komercyjnego oprogramowania analizującego proces planowania obciążenia. Niewiele firm przeprowadza tego typu planowanie obciążenia. Są to bardzo duże przedsiębiorstwa dysponujące budżetem wystarczającym do tego, żeby sobie pozwolić na takie dokładne planowanie (firmy te decydują się na nie głównie dlatego, że ostatecznie poniesione koszty zwracają się).

Największą różnicą między anegdotycznym planowaniem obciążenia i planowaniem go w przedsiębiorstwie jest zakres działań. Pierwsza z wymienionych metod bazuje na praktycznym podejściu i bardziej na założeniach popartych wiedzą, natomiast planowanie obciążenia w przedsiębiorstwie to dogłębne studium wymagań i wydajności, którego celem jest uzyskanie jak najbardziej precyzyjnych wartości.

Planowanie obciążenia serwera Tomcat

W celu zaplanowania obciążenia serwerów z uruchomionym Tomcatem można przeanalizować i przewidzieć dowolną z poniższych pozycji (lista nie ma pełnić roli kompletnej, lecz wyszczególniać kilka typowych pozycji).

Sprzęt komputera pełniącego rolę serwera
: Jakie zastosować architektury komputerowe? Ile komputerów będzie wymaganych przez witrynę? Czy może jeden bardzo wydajny, czy wiele wolniejszych? Ile procesorów zainstalować w każdym komputerze, a ile pamięci RAM? Ile będzie potrzebnej przestrzeni na dyskach twardych i jak szybkie muszą być układy wejścia-wyjścia? Jak będzie wyglądać bieżąca konserwacja sprzętu? Jak użycie innych implementacji wirtualnej maszyny Java wpłynie na wymagania sprzętowe?

Przepustowość sieci
: Jaka w godzinach szczytu będzie wymagana przepustowość połączeń sieciowych obsługujących przychodzący i wychodzący ruch sieciowy? W jaki sposób można zmodyfikować aplikację WWW w celu zmniejszenia tych wymagań?

System operacyjny serwera
: Jaki system operacyjny najlepiej sprawdzi się w przypadku witryny udostępniającej dane? Jakie implementacje maszyny JVM są dostępne dla każdego systemu i w jakim stopniu każda z nich wykorzystuje możliwości systemu operacyjnego? Czy na przykład wirtualna maszyna Java we własnym zakresie obsługuje wielowątkowość? A co z wieloprocesorowością symetryczną SMP (*Symmetric MultiProcessing*)? Jeśli jest ona obsługiwana przez maszynę JVM, czy powinno się rozważyć użycie serwerów z wieloma procesorami? Jaka konfiguracja sprzętowa będzie obsługiwać aplikacje WWW szybciej, bardziej niezawodnie i przy mniejszych kosztach — wiele serwerów 1-procesorowych czy pojedynczy serwer z 4 procesorami?

Oto ogólna procedura przeznaczona dla każdego typu planowania obciążenia, a zwłaszcza dla planowania obciążenia w przypadku serwera Tomcat:

1. Opis obciążenia roboczego. Jeśli witryna już funkcjonuje, można określić liczbę żądań przetwarzanych w ciągu sekundy, zestawić różnego rodzaju możliwe żądania i zmierzyć poziom wykorzystania zasobów z podziałem na typ żądania. Jeżeli witryna nie została jeszcze uruchomiona, w celu określenia wymagań dotyczących zasobów, korzystając z posiadanej wiedzy, można założyć liczbę żądań i przeprowadzić testy przygotowawcze.

2. Analiza trendów wydajnościowych. Trzeba wiedzieć, jakie żądania generują największe obciążenie i jak pod tym względem wypadają innego typu żądania. Dysponowanie wiedzą na temat tego, jakie żądania powodują największe obciążenie lub zużywają najwięcej zasobów, ułatwi identyfikację tego, co należy zoptymalizować, żeby wynik operacji miał jak najlepszy ogólny wpływ na serwery. Jeśli na przykład serwlet kierujący zapytanie do bazy danych zbyt długo odsyła odpowiedź, być może przez buforowanie części danych w pamięci RAM bezpiecznie skróci się czas oczekiwania.

3. Podjęcie decyzji odnośnie do minimalnych dopuszczalnych wymagań usługi. Przykładowo można nie chcieć, żeby użytkownik kiedykolwiek czekał dłużej niż 20 sekund na odpowiedź wysyłaną przez witrynę internetową. Oznacza to, że nawet podczas szczytowego obciążenia dla żadnego żądania całkowity czas upływający od momentu zainicjowania żądania do chwili otrzymania odpowiedzi nie przekroczy 20 sekund. Może to uwzględniać wszelkie zapytania bazodanowe i operacje dostępu do systemu plików niezbędne do zrealizowania żądania aplikacji, która najintensywniej korzysta z zasobów. Minimalne dopuszczalne wymagania usługi są specyficzne dla poszczególnych firm i się zmieniają. Do innego rodzaju minimalnych wymogów dotyczących usług należy zaliczyć najmniejszą liczbę żądań, które w ciągu sekundy witryna musi obsłużyć, a także minimalną liczbę jednoczesnych sesji i użytkowników.
4. Zdecydowanie o tym, jakie zostaną użyte zasoby infrastruktury i przetestowanie ich w środowisku przygotowawczym. Do zasobów tych należy zaliczyć sprzęt komputerowy, połączenia sieciowe, oprogramowanie systemu operacyjnego itp. Należy zamówić, wdrożyć i przetestować przynajmniej jeden serwer, który jest kopią lustrzaną serwera produkcyjnego, oraz sprawdzić, czy spełnia postawione wymagania. Testując serwer Tomcat, trzeba pamiętać o wypróbowaniu więcej niż jednej implementacji wirtualnej maszyny Java, sprawdzeniu różnych ustawień rozmiaru pamięci i puli wątków żądań (była o tym mowa wcześniej w rozdziale).
5. Jeżeli w kroku 4. zostały spełnione wymagania usługi, można zamówić i wdrożyć więcej egzemplarzy tego, co będzie pełnić rolę serwera produkcyjnego. W przeciwnym razie krok 4. należy powtarzać do momentu spełnienia wymagań stawianych przez usługę.

Trzeba udokumentować wyniki działań, ponieważ tworzą one czasochłonny proces, który musi być powtórzony, gdy ktoś będzie chciał się dowiedzieć, jak firma doszła do określonych wniosków. Poza tym ze względu na to, że testowanie jest operacją powtarzalną, istotne jest dokumentowanie wszystkich wyników każdego cyklu testów, a także ustawień konfiguracyjnych powodujących uzyskanie takich wyników. Dzięki temu będzie można stwierdzić, kiedy dostrajanie nie daje już zauważalnie pozytywnych rezultatów.

Po zakończeniu planowania obciążenia witryna będzie znacznie lepiej zoptymalizowana pod kątem wydajności, przede wszystkim na skutek rygorystycznych testów różnych opcji. Powinno się uzyskać znaczący wzrost wydajności tylko dzięki doborowi odpowiedniej kombinacji sprzętu, systemu operacyjnego i wirtualnej maszyny Java dla konkretnego przypadku zastosowania serwera Tomcat.

Dodatkowe źródła informacji

Jak wcześniej wspomniano, jeden rozdział ledwie wystarcza do szczegółowego omówienia dostrajania wydajności. Wskazane są dodatkowe poszukiwania, analiza optymalizowania aplikacji Java, dostrajanie systemów operacyjnych, uzyskanie informacji na temat przeprowadzania planowania obciążenia w przypadku wielu serwerów i aplikacji, a także wszelkie inne działania mające związek z konkretnym zastosowaniem. Na początek można skorzystać z kilku źródeł informacji, które w naszym przypadku okazały się pomocne.

W książce *Java Performance Tuning* autorstwa Jacka Shiraziego (wydawnictwo O'Reilly) omówiono wszystkie kwestie dotyczące dostrajania aplikacji Java; zamieszczono w niej dobry materiał poświęcony wydajności wirtualnej maszyny Java. Jest to znakomita publikacja, w której

zawarto bardzo szczegółowe informacje na temat kwestii wydajnościowych na etapie projektowania aplikacji. Oczywiście, serwer Tomcat jest aplikacją Java, dlatego też spora część tego, o czym Jack pisze w swojej książce, dotyczy wersji Tomcata, którą Czytelnik zainstaluje. Jak już wspomniano wcześniej w rozdziale, przez samą edycję plików konfiguracyjnych Tomcata na kilka sposobów można poprawić wydajność.

Trzeba pamiętać, że choć serwer Tomcat jest oprogramowaniem *open source*, zalicza się również do bardzo złożonych aplikacji. W związku z tym, wprowadzając zmiany w jego kodzie źródłowym, należy zachować ostrożność. Korzystając z list wysyłkowych poświęconych Tomcatowi, można dzielić się z innymi własnymi pomysłami, a także zaangażować się w działania społeczności użytkowników związanych z tym serwerem, gdy postanowi się dokładniej poznać jego kod źródłowy.

Jeżeli uruchomiono witrynę internetową o tak dużym natężeniu ruchu sieciowego, że jeden serwer może nie być wystarczający do obsłużenia całego obciążenia, prawdopodobnie powinno się przeczytać rozdział 10. Omówiono w nim uruchomienie witryny jednocześnie w obrębie więcej niż jednej instancji Tomcata, z możliwością wykorzystania kilku komputerów pełniących rolę serwera.

Aby znaleźć więcej stron internetowych poświęconych planowaniu obciążenia, w wyszukiwarce wystarczy wprowadzić termin `planowanie obciążenia` lub `capacity planning`. Kilka dobrych przykładów związanych z tym zagadnieniem można znaleźć pod adresami *http://en.wikipedia.org/*
↪wiki/Capacity_planning i *http://www.informit.com/articles/article.asp?p=27641&rl=1*.