Implementacja

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

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

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

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

Kod modułu był pisany pod kątem maksymalizowania wydajności. Ponieważ zazwyczaj w tego typu systemach wąskim gardłem staje się przepustowość sieci, główna część uwagi podczas tworzenia projektu była poświęcona kwestii minimalizowania ilości przesyłanych informacji. Wiąże się to z wyeliminowaniem zbędnych pakietów (por. p. 4.3.1, 4.3.2) oraz z mechanizmami buforowania (por. p. 4.3.4). Dodatkowym utrudnieniem podczas pisania było silnie zrównoleglone środowisko pracy modułu. Serwer WWW pracuje wielowątkowo, obsługując nawet kilkaset

żądań jednocześnie. Dobrze napisany moduł klastra musi być zabezpieczony przed równoległym dostępem, jednocześnie pozwalając na wykonywanie możliwie jak największej liczby operacji w tym samym czasie. Zbyt mała liczba rozłącznych sekcji krytycznych spowolni działanie modułu, tworząc wąskie gardła. Z kolei zbyt częste wywołania wejścia i wyjścia z sekcji krytycznej może negatywnie wpływać na wydajność i utrudniać wykrywanie błędów.

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

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

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

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

org.apache.catalina.cluster.

Warstwa transportująca

Ponieważ nowy klaster ma być rozwiązaniem tanim i łatwym w instalacji, jako mechanizm transportujący został wykorzystany protokół TCP/IP, przy użyciu którego komputery mogą się komunikować niemalże w każdej dostępnej sieci. Drugim bardzo ważnym czynnikiem, który spowodował wybór tej technologii jest powszechność gotowych, sprawdzonych implementacji (obsługa

TCP/IP jest zaimplementowana w standardowych klasach Javy). Charakterystyka architektury klastra (połączenia każdy z każdym) sugerowałaby wykorzystanie protokołu rozgłoszeniowego, niemniej jednak brak mechanizmów gwarantujących dostarczenie danych oraz sprawdzenia ich poprawności, spowodował wykluczenie tego rozwiązania z projektu. Do poprawności działania klastra niezbędna jest gwarancja dostarczenia danych w niezmienionej postaci. Taką gwarancję daje protokół TCP/IP.

Na implementację warstwy transportującej składają się następujące klasy:

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

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

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

Obowiązkiem obiektu DestinationManager jest obsługa wszelkich problemów związanych z połączeniem i przesyłaniem danych.

Sporą wadą protokołu TCP/IP jest łatwość zerwania połączenia – każde połączenie jest uznawane za aktywne dopóki są otrzymywane pakiety.

Niestety, wystarczy, że z powodu przeciążenia sieci lub systemu operacyjnego pakiety zaczną się znacząco opóźniać i połączenie może zostać zerwane. Ponieważ w zastosowanej architekturze spójność klastra jest kontrolowana za pomocą aktywności połączeń, aby ograniczyć liczbę omyłkowych decyzji o awarii węzłów, implementacja warstwy transportującej umożliwia przezroczyste odtwarzanie chwilowo zerwanych połączeń. Po utracie połączenia system przez pewien czas próbuje odtworzyć kanał komunikacyjny; jeżeli się to nie powiedzie, to informuje wszystkich zarejestrowanych słuchaczy o utracie adresata.

Mechanizm powiadamiania o utracie połączenia jest wykorzystywany przez klaster do podejmowania decyzji o awarii węzłów. W przypadku zwykłego węzła znaczenie ma tylko informacja o zerwaniu połączenia z serwerem zarządcą – węzeł musi zerwać wszystkie pozostałe połączenia i próbować odnowić komunikację z zarządcą. Z kolei w przypadku serwera zarządcy informacja o zerwaniu połączenia powoduje podjęcie decyzji o jego odrzuceniu z klastra.

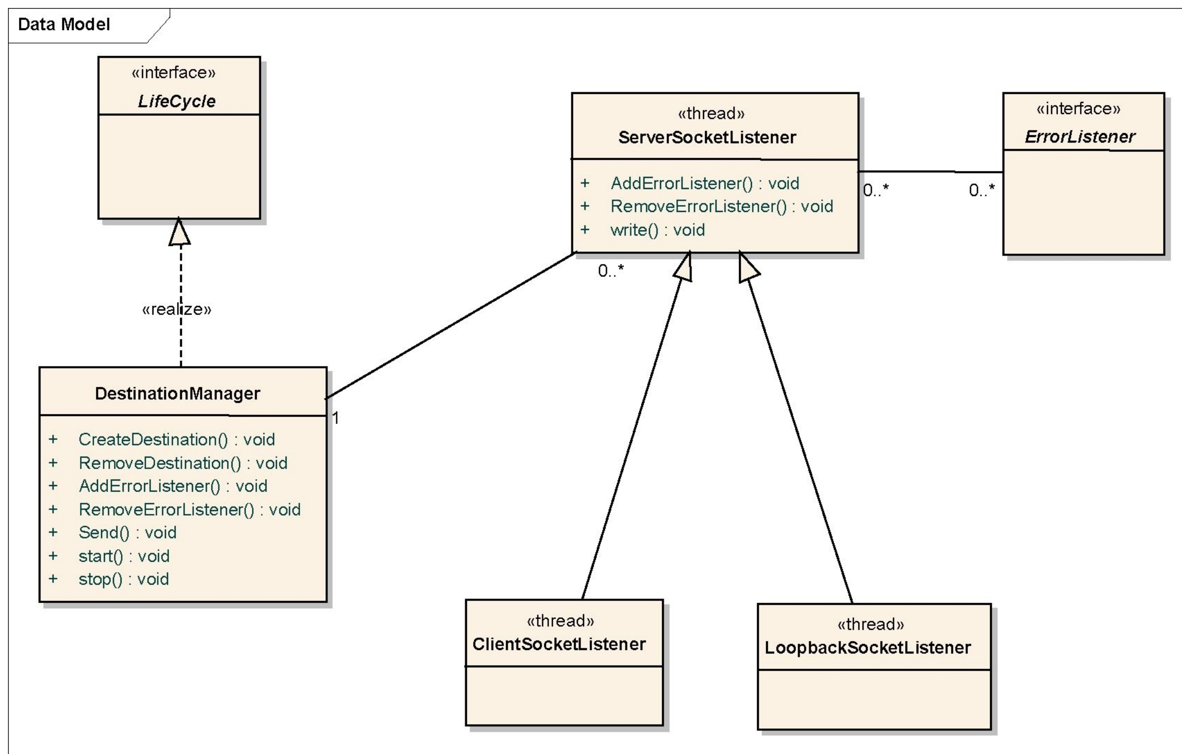
Ważną cechą obiektu klasy DestinationManager jest fakt, że dany adresat będzie uznawany przez obiekt jako aktywny dopóki nie zostanie sxplicits wywołane RemoveDestination (DestinationAddress). Ten fakt ułatwia implementację mechanizmu replikacji – węzeł wysyła dane do wszystkich pozostałych węzłów, niezależnie od tego czy są one dla niego dostępne czy nie. Obiekt DestinationManagei po prostu będzie bez końca ponawiał próbę wysłania lub stwierdzi fakt poprawnego zakończenia, jeżeli w międzyczasie adresat zostanie usunięty z listy aktywnych.

Aby uniknąć aktywnego oczekiwania na grupie połączeń, każdy kanał komunikacyjny ma swój własny wątek, który go obsługuje.

Istnieją trzy typy połączeń: ServerSocketListene: , ClientSocketListener oraz

LoopbackSocketListener. Diagram na rys. 4.8 przedstawia hierarchię klas.

Rys. 4.8. Hierarchia klas warstwy transportującej



W celu zminimalizowania liczby przesyłanych pakietów w sieci, pomiędzy każdymi dwoma węzłami otwierane jest tylko jedno połączenie. Wyróżnia się stronę bierną i aktywną. Strona bierna, czyli obiekt klasy ServerSocketListene , nasłuchuje na porcie identyfikującym węzeł, oczekując na zainicjowanie połączenia przez stronę aktywną. Strona aktywna, czyli obiekt klasy clientSocketListene , nawiązuje komunikację łącząc się na odpowiedni port adresata.

Ponieważ strona bierna nie może zidentyfikować strony aktywnej (nie jest w stanie poznać numeru portu, na którym nasłuchuje strona aktywna, a jedynie adres IP, z którego się łączy), więc węzeł inicjujący połączenie wysyła w pierwszej kolejności numer swojego portu. Rola w połączeniu jest ustalana na podstawie wartości, którą przekaże metoda compareTo

(DestinationAddress). Poniższy kod jest wykonywany podczas tworzenia połączenia W obiekcie DestinationManager:

if (oAddr.compareTo(oLocalAddr) < 0)

oCL = new ServerSocketListener(oAddr, this); else if (oAddr.compareTo(oLocalAddr) > 0)

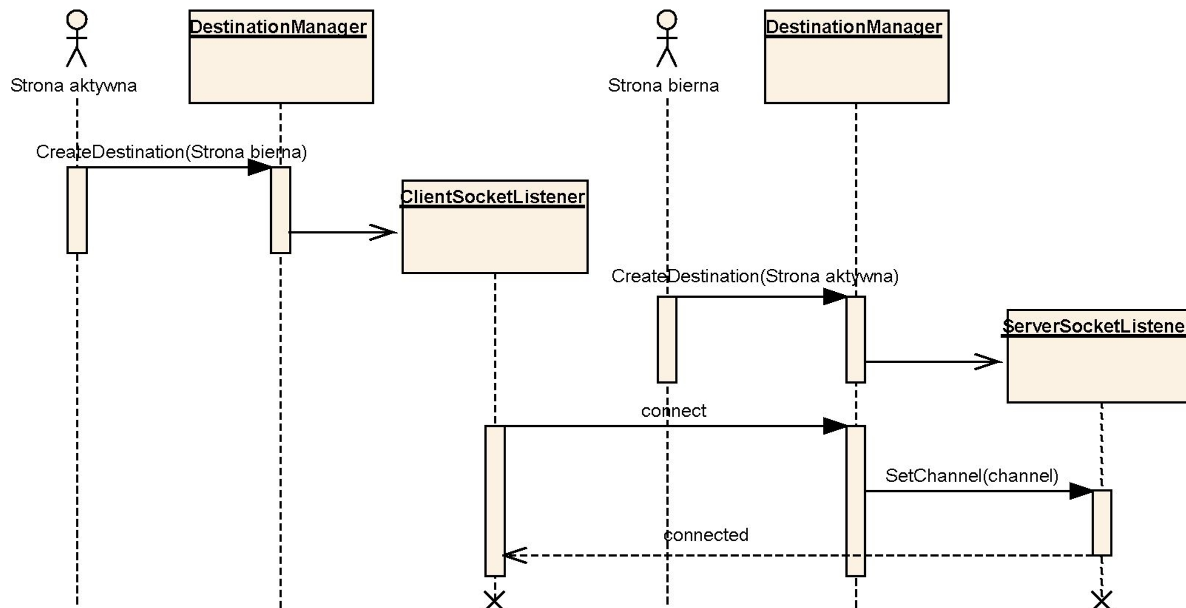
oCL = new ClientSocketListener(this.oLocalAddr, oAddr, this); else

throw new RuntimeException(„Will not add localhost”);

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

Schemat procesu nawiązywania połączenia jest przedstawiony na rys. 4.9.

Rys. 4.9. Diagram nawiązywania połączenia



Obiekt DestinationManagei podczas tworzenia otwiera port do nasłuchu. W momencie próby nawiązania połączenia (powrót z metody accept ()), menedżer sprawdza czy istnieje obiekt konektora obsługujący adres zgłaszającego się komputera. Jeżeli istnieje, to wywołuje na nim metodę Setchannel (socket , tym samym kończąc procedurę nawiązywania połączenia. Jeżeli nie istnieje, to wywołuje na obiekcie klasy '4ainSyncServer (dokładniej o tej klasie będzie mowa w p. 4.3.6), metodę canAccept (DestinationAddress). Jeżeli metoda przekaże wartość pozytywną, to menedżer wywołuje sam dla siebie metodę CreateDestination z nowym adresatem jako argumentem. W tym momencie cała procedura nawiązywania połączenia zaczyna się na nowo. Taki mechanizm umożliwia serwerowi zarządcy akceptowanie połączeń z nowymi, nieznanymi wcześniej węzłami.

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

Poza dwoma podstawowymi typami połączeń zostało jeszcze

stworzone połączenie zwrotne, obiekt klasy LoopbackSocketListener. Wszystkie wysyłane przez niego dane od razu przekierowywane są do funkcji obsługi wiadomości sieciowych, nie wywołując niepotrzebnie funkcji systemowych. Jest on wykorzystywany podczas przesyłania danych od serwera zarządcy do serwera Tomcat w przypadku, gdy serwer Tomcat jest jednocześnie serwerem zarządcą. Architektura rozwiązania wymusiła taki mechanizm – serwer zarządca nie wie nic o istnieniu serwera Tomcat. Jego zadanie sprowadza się jedynie do odbierania i wysyłania odpowiednich komunikatów. Takie podejście umożliwia odseparowanie serwera zarządcy od Tomcata (por. p. 4.3.7).

Odbieranie danych z sieci

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

AddReadyBuffer(DestinationAddress oFromAddr, ExtendableByteBuffer oData) .

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

Implementacja warstwy transportującej jest bardzo silnie zależna od protokołu TCP/IP. Przenośność i uogólnianie części transportującej nie były głównymi czynnikami decydującymi o kształcie końcowego kodu. Najważniejsza była wydajność tej części modułu, ponieważ to od niej zależała wydajność całego projektu. Takie zabiegi jak tworzenie pojedynczych połączeń,

osobne wątki dla każdego kanału komunikacyjnego czy kolejki buforów miały na celu optymalizację modułu klastra. Mechanizm odtwarzania połączeń miał na celu zabezpieczenie klastra przed przypadkowym rozbiciem oraz ułatwienie implementacji samego mechanizmu replikowania sesji.

Komunikacja w klastrze

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

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

Typy wiadomości można podzielić na dwa zbiory:

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

Wiadomości przesyłane między dwoma zwykłymi węzłami sessions_add – dodanie nowej lub modyfikacja istniejącej sesji. W sekcji danych znajduje się identyfikator sesji, kontekst oraz zserializowane dane sesji.

Wiadomości przesyłane między zwykłym węzłem i serwerem zarządcą Wiadomości przesyłane od serwera zarządcy do zwykłego węzła:

- destinations_ade – dodanie nowego węzła do klastra. W sekcji danych przesyłany jest adres nowego węzła.

- destination_remove – usunięcie węzła z klastra. W sekcji danych przesyłany jest adres węzła.
- obtained_session – sesja została przyznana węzłowi. W sekcji danych znajduje się identyfikator sesji, kontekst oraz aktualny numer wersji (jeżeli węzeł posiada mniejszy numer oznacza to, że z jakiś powodów nie otrzymał repliki ostatnich zmian i musi wstrzymać modyfikację sesji do czasu otrzymania najświeższych danych).
- session_existance – informacja czy podana sesja istnieje. W sekcji danych znajduje się identyfikator sesji, kontekst oraz 1 bajt z informacją czy sesja istnieje w klastrze czy nie. Komunikat jest wysyłany w odpowiedzi na

CHECK_SESSION_EXISTANCE.

- check_session_version – prośba o odesłanie numeru wersji sesji o podanym identyfikatorze. W sekcji danych przesyłany jest identyfikator sesji oraz kontekst. Komunikat jest wysyłany w momencie próby odzyskania sesji zajmowanych przez węzeł, który doznał awarii. Każdy węzeł w klastrze odsyła numer wersji, a serwer zarządca na tej podstawie wybiera węzeł, który zreplikuje swoją kopię danych do pozostałych członków.
- s e s s i on s_remove – usunięcie sesji. W sekcji danych znajduje się tablica identyfikatorów sesji oraz kontekstów. Komunikat jest wysyłany w momencie wygaśnięcia sesji. Serwer zarządca podejmuje decyzję o zakończeniu życia sesji analogicznie do pojedynczego serwera Tomcat. Jeżeli przez odpowiednio długi okres żaden serwer nie poprosi o dostęp do sesji, to zostaje ona uznana za nieaktywną.
- sessions_replicate_with_sync – wymuszenie procesu replikacji. W sekcji danych znajduje się adres węzła, do którego należy wysłać replikę (jeżeli jest pusty, to należy rozesłać repliki do wszystkich węzłów) oraz

tablica identyfikatorów sesji, które należy replikować. Węzeł, który odbierze komunikat po zakończeniu replikacji musi zwolnić wszystkie sesje, ponieważ serwer zarządca przed wysłaniem wiadomości ustawia blokady na adresata. Mechanizm ten zabezpiecza przed niebezpieczeństwem wprowadzania zmian w sesjach przez inny węzeł w czasie replikowania. Wysłane repliki mogłyby nadpisać nowo wprowadzone zmiany.

Wiadomości przesyłane od zwykłego węzła do serwera zarządcy:

- obtain_session – zajęcie sesji. W sekcji danych znajduje się identyfikator zajmowanej sesji oraz kontekst.
- release_session – zwolnienie sesji. W sekcji danych znajduje się identyfikator zwalnianej sesji, kontekst, numer wersji sesji oraz liczba zwolnień (zazwyczaj 1). Może się zdarzyć, że węzeł posiadając blokadę na sesji otrzyma od serwera zarządcy komunikat sessions_replicate_with_sync (na przykład z powodu zgłoszenia się nowego węzła do klastra). Wtedy węzeł musi zwolnić sesję więcej niż raz, wysyłając w tym celu liczbę zwolnień.
- new_session_member – zgłoszenie się nowego węzła w klastrze. W sekcji danych znajduje się słownik par: identyfikator sesji (wraz z kontekstem), numer wersji. Komunikat jest wysyłany za każdym razem, gdy nastąpi zerwanie połączenia między zwykłym węzłem a serwerem zarządcą. W przypadku zupełnie nowego węzła słownik będzie pusty.
- session_created – utworzenie sesji. W sekcji danych znajduje się identyfikator sesji oraz kontekst. Komunikat jest wysyłany zaraz po utworzeniu sesji. Wysłanie komunikatu jest konieczne, aby pozostałe węzły mogły się dowiedzieć poprzez komunikat check_session_existance o istnieniu tej sesji zanim twórca zakończy proces replikacji.
- check_session_existance – sprawdzenie istnienia sesji. W

sekcji danych znajduje się identyfikator sesji oraz kontekst. Komunikat jest wysyłany, jeżeli węzeł otrzyma żądanie odwołujące się do nieistniejącej sesji. Ponieważ sesja mogła być stworzona na innym węźle, ale jeszcze nie doszła replika, serwer wpierw sprawdza u zarządcy czy identyfikator jest aktywny.

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

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

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

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

Mechanizm kolejki zadań zależnych

Na potrzeby projektu została stworzona wyspecjalizowana kolejka do przetwarzania zadań. Kolejka posiada swoją własną pulę wątków przetwarzających instrukcje. W ten sposób istnieje możliwość zlecania zadań, które mają się wykonać

asynchronicznie (jak np. replikacja sesji). Wątki tworzone są raz, w momencie tworzenia kolejki, a ich praca sprowadza się do oczekiwania na nadejście kolejnego zadania, które mogłyby wykonać. Jeżeli zadań nie ma, to wątki przechodzą w stan uśpienia. Zmieniając liczbę wątków w kolejce można kontrolować zużycie zasobów systemowych przez moduł klastra (szerzej o konfigurowaniu klastra będzie mowa w p. 4.4).

Kolejka zadań posiada dodatkowe, bardzo przydatne możliwości:

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

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

org..task.Listener:

```
public interface Listener {

public void WakeUp(boolean bError);

}
```

W momencie zakończenia wykonywania zadania dla każdego dołączonego słuchacza wywoływana jest metoda wakeup (boolean bError), z argumentem informującym czy zadanie zakończyło się z błędem czy nie (błąd jest wynikiem zgłoszenia przez zadanie

wyjątku). Ten mechanizm jest wykorzystywany przy implementacji zależności między zadaniami. Po prostu klasa Task implementuje interfejs słuchacza, a w metodzie WakeUp (boolean) zmniejsza licznik zadań, na które oczekuje. W momencie, gdy licznik spadnie do zera, zadanie samo się inicjuje, dodając się do kolejki:

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

}

public synchronized void DecTasks() { nTasks–;

Start();

}

public void Start()     {

if (nTasks <= 0)       {

if (this.nTimeoutFromStart > 0)

this.SetTime(System.currentTimeMillis()              +

nTimeoutFromStart); oTaskQueue.AddTask(this);

}

}
```

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

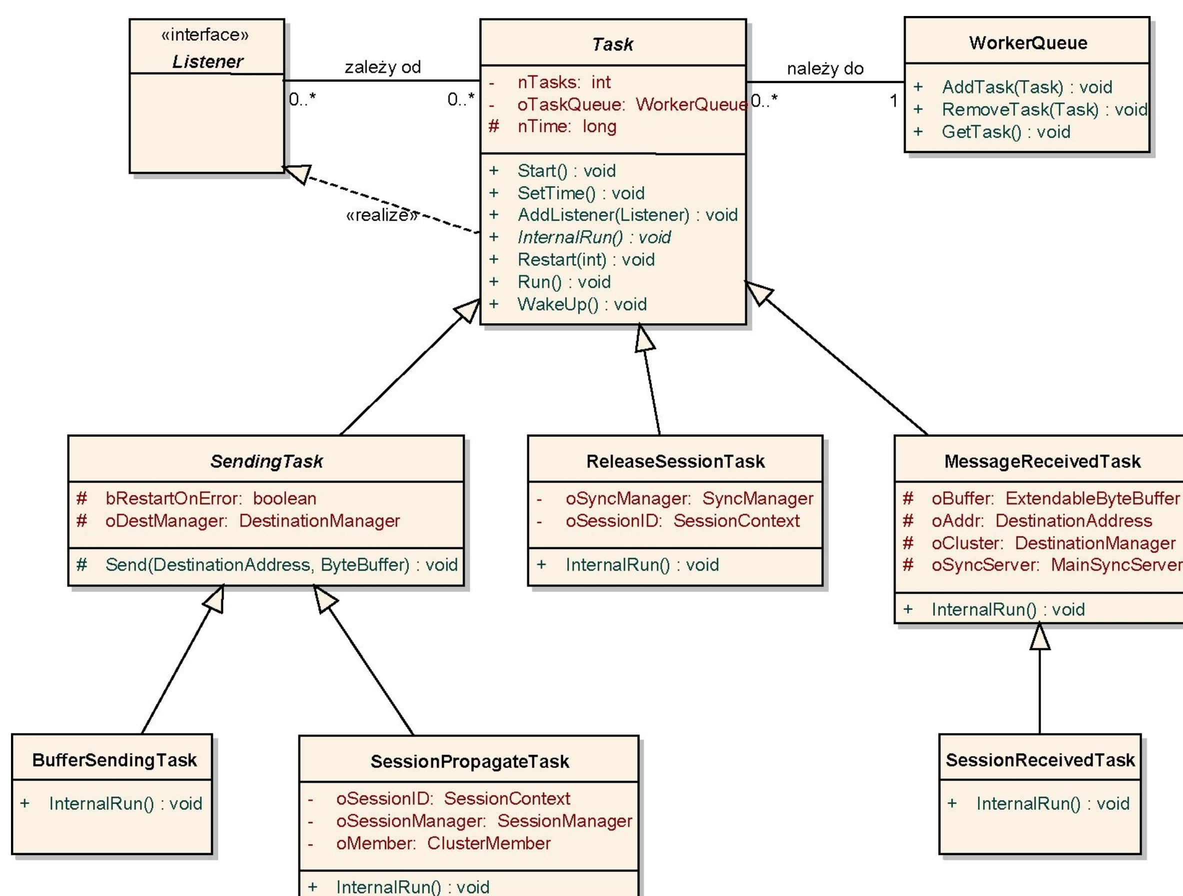
```
public void WakeUp(boolean bError) { if (! bError)

super.WakeUp(bError) ;
```

}

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

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

Rys. 4.10. Diagram hierarchii zadań



Opis zadań:

1. sendingTask – abstrakcyjna klasa, zawierająca metodę Send (DestinationAddress, ByteBuffer), która umożliwia wysłanie bufora z danymi do konkretnego adresata. Można

skonfigurować zadanie w dwóch różnych trybach:

2. Operacja wysyłania ma być blokująca – tzn. powrót z metody 3end () nastąpi dopiero po faktycznym wysłaniu danych.
3. Błąd podczas wysyłania ma powodować restart zadania.
4. BufferSendingTast – klasa w metodzie internaiRun () wywołuje metodę send () z nadklasy. Klasa jest wykorzystywana do generowania zadań wysyłających proste komunikaty (np. komunikat o typie j bt ai u se s s i oh).
5. sessionPropagateTask – klasa jest odpowiedzialna za wysłanie repliki konkretnej sesji do konkretnego adresata. W metodzie InternaiRun () serializuje obiekt sesji i próbuje wysłać dane do węzła (wywołując metodę Send () z nadklasy). Klasa wykorzystuje tryb restartowania w przypadku błędu. W ten sposób, jeżeli z jakiegoś powodu wysłanie danych zaczyna się opóźniać, to nie blokuje niepotrzebnie pamięci zserializowanymi danymi sesji i co ważniejsze przy kolejnym uruchomieniu zadania ponawia próbę replikacji, ale już być może z nowszą wersją sesji, zmniejszając w ten sposób liczbę przesyłanych informacji (tworzenie zadań replikacyjnych zostanie opisane w p. 4.3.5). Ponadto, jeżeli system wykorzystuje buforowanie, to przy definicji zadania ustala się opóźnienie w wykonaniu (por. p. 4.3.4).
6. ReieaseSessionTask – zadania tej klasy są odpowiedzialne za zwalnianie sesji w serwerze zarządcy. System podczas tworzenia ustawia zależność tego zadania od wszystkich zadań replikacyjnych sesji. Proces zwolnienia jest uruchamiany dopiero po poprawnym przesłaniu replik do wszystkich węzłów w klastrze. W przypadku wykorzystywania buforowania system dodatkowo opóźnia start zadania już po wykonaniu wszystkich replikacji (szerzej na ten temat będzie mowa w p. 4.3.4).
7. MessageReceivedTask – zadanie tej klasy tworzone jest po odebraniu wiadomości z sieci. Zadanie zawiera dane komunikatu oraz adres węzła, który je przysłał.

Przetwarzanie wiadomości wykonywane jest w odrębnym wątku, co powoduje, że nie następuje blokowanie wątku odbierającego dane z sieci. W ten sposób zapewniane jest maksymalne zrównoleglenie obliczeń wykonywanych przez moduł klastra, co może nie być bez znaczenia w przypadku serwerów wieloprocesorowych.

8. SessionReceivedTask – klasa dziedziczy po MessageReceivedTask dodając w metodzie internaiRun () obsługę komunikatów charakterystycznych w sytuacji, gdy moduł jest zanurzony wewnątrz serwera Tomcat, jak np. komunikat

W systemie zostały stworzone dwie odrębne kolejki zadań:

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

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

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

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

Implementacja kolejki zadań

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

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

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

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

Buforowanie

W systemie zostało zastosowane buforowanie w dwóch różnych etapach:

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

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

Opóźnianie momentu replikacji sesji

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

Takie podejście gwarantuje, że nawet jeżeli wiele wątków jednocześnie będzie przetwarzało żądania odwołujące się do tej samej sesji, to tak czy owak zostanie stworzone tylko jedno zadanie replikujące, zmniejszając w ten sposób liczbę przesyłanych informacji. Dodatkowo mechanizm pozwala na łatwą realizację koncepcji opóźniania procesu replikacji. Wystarczy, że podczas tworzenia zadania replikującego moduł wstawi opóźnienie w jego wykonanie. Jeżeli w czasie oczekiwania zostanie przetworzone kolejne żądanie (zmieniające dane

sesji), to moduł ominie wysłanie wcześniejszej wersji. W ten sposób doprowadza się do sytuacji, gdzie wydajność klastra przestaje zależeć od częstotliwości zgłaszania się użytkowników. W rezultacie sesja będzie replikowane co pewien czas, niezależnie od tego czy użytkownik zgłosi się do systemu raz czy bardzo wiele razy. Dobór długości opóźnienia nie jest rzeczą zupełnie trywialną – jeżeli będzie za małe, to buforowanie może nie przynosić oczekiwanych rezultatów. Z kolei jeżeli opóźnienie będzie za duże, to system straci swoją główną zaletę – czyli dynamiczne równoważenie obciążenia. Doprowadzi to do sytuacji, gdzie każda sesja jest na stałe przypisana do konkretnego węzła – czyli analogicznie do koncepcji zastosowanej w serwerze Bea Weblogic 8.0 (por. p. 2.3.2.1), a cały nakład związany z synchronizowaniem sesji okaże się zbędny.

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

CHECK_SESSION_VERSION oraz SESSION_VERSION W p. 4.3.2).

Opóźnianie momentu zwalniania sesji

Poza opóźnianiem wysłania pakietów z kopią najnowszej wersji sesji można również opóźniać moment zwolnienia sesji. Wydajność klastra jest mierzona szybkością reakcji systemu na zgłoszenie użytkownika. Im reakcja jest szybsza, tym lepiej.

Niestety architektura zaimplementowanego klastra wymaga, aby przed modyfikacją danych użytkownika system zablokował sesję w serwerze zarządzającym. Operacja ta jest dosyć kosztowna – wymaga przesłania dwóch pakietów w sieci (prośba o zablokowanie i odpowiedź). Jeżeli węzeł opóźni moment oddania sesji, to może przetworzyć kilka kolejnych żądań bez potrzeby ponownego zajmowania sesji. Oczywiście nie można przesadzić z długością opóźnienia, aby nie doprowadzić do sytuacji opisanej w poprzednim punkcie, gdy klaster zaczyna się zachowywać tak, jakby istniały stałe przypisania sesja – węzeł macierzysty.

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

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

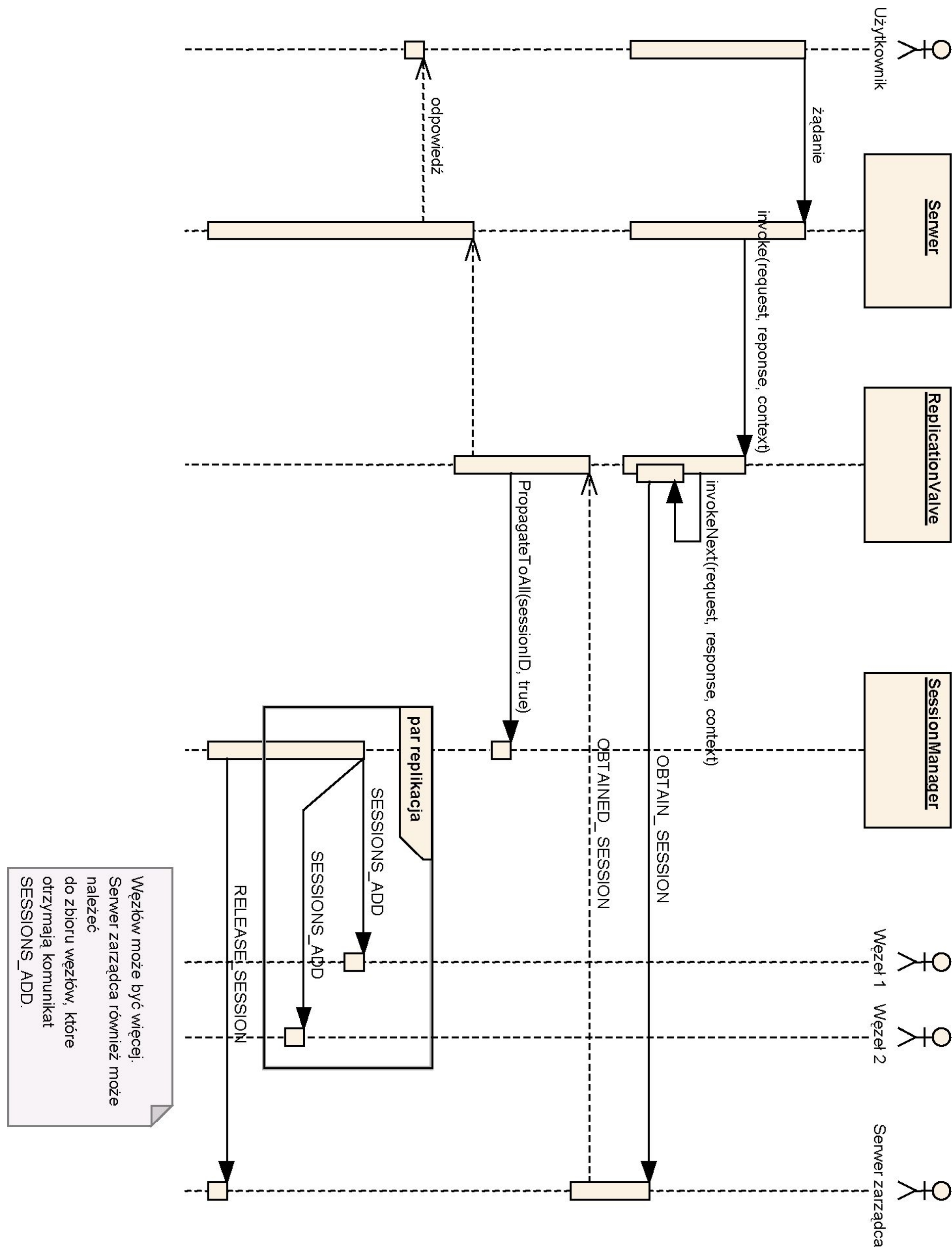
Algorytm przetwarzania żądania

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

Na rys. 4.11 znajduje się diagram przepływów ukazujący

najbardziej charakterystyczny przypadek obsługi żądania. Diagram przedstawia zgłoszenie klienta z numerem sesji, którą należy zablokować w serwerze zarządzającym.

Rys. 4.11. Diagram przepływ u podczas obsługi żądania



Klient generuje zgłoszenie, które zostaje wysłane do serwera

Tomcat. Serwer inicjuje wykonanie strumienia przetwarzania żądań (por. p. 3.3.2). Wśród załączonych wyzwalaczy występuje również wyzwalacz klasy ReplicationValve (analogicznie do implementacji Filipa Hanika). W metodzie invoke(Request, Response, Context) wyzwalacz wznawia przetwarzanie strumienia w celu wykonania Zgłoszenia. Po powrocie Z metody invokeNext (Reąuest, Response, Context) program sprawdza czy żądanie posiada sesję oraz czy sesja jest dzielona przez cały klaster. Jeżeli tak, to zwiększa wersję sesji, tworzy dla każdego węzła w klastrze zadanie replikacyjne oraz tworzy zadanie odblokowania sesji, które wykona się po rozesłaniu wszystkich kopii. Następnie kończy działanie, tym samym kończąc proces przetwarzania żądania. Wszystkie stworzone zadania wykonane zostaną asynchronicznie przez wątki obsługujące kolejkę zadań.

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

Proces tworzenia sesji

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

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

Proces sprawdzania istnienia sesji

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

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

O ile w pierwszym przypadku menedżer przez pewien czas może

zachowywać informacje o wygasłych sesjach i bez potrzeby komunikowania się z zarządcą po prostu tworzyć nową sesję, o tyle w drugim przypadku konieczne jest wysłanie zapytania.

Obiekt klasy SessionManager w metodzie findSession(String sessionID)

wykonuje następujący kod:

public Session findSession(String sSessionID) throws java.io.IOException { return findSession(sSessionID, true);

}

public Session findSession(String sSessionID, boolean bCreate) throws java.io.IOException { if (sSessionID == null) return null;

Session oSession = super.findSession(sSessionID); if (oSession == null && sSessionID != null) {

// sprawdzamy czy sesja istnieje w klastrze

if (oSyncServer.CheckExistance(new SessionContext(sSessionID, this.getName()))) {

synchronized (sessions) {

oSession = super.findSession(sSessionID);

if (oSession == null) {

// sesja istnieje, ale my wciąż jej nie mamy // dlatego tworzymy pustą, aby wątek mógł // przy odwołaniu rozpocząć procedurę // blokowania sesji. oSession =

this.createSession(sSessionID, false);

}

}

```
}

return oSession;
```

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

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

Proces blokowania sesji poprzez odwołanie

W celu zminimalizowania niepotrzebnych operacji blokowania oraz replikowania sesji moment wejścia do sekcji krytycznej został przeniesiony w miejsce faktycznego odwołania do danych sesji. Czyli jeżeli użytkownik wygeneruje żądanie, które nie

będzie zaglądało do danych sesji (nie zostaną wywołane na sesji metody

getAttribute(String), setAttribute(String, Object) itp.), to nie spowoduje

to żadnego ruchu po stronie klastra.

W dalszej części tego punktu opisano sposób zaimplementowania tego mechanizmu.

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

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

Tak czy owak w najgorszym wypadku węzeł po prostu nie zwolni sesji, co przy wykorzystaniu zintegrowanego serwera ruchu nie spowoduje zaprzestania działania aplikacji. Wszystkie żądania

będą kierowane na jedną maszynę bez możliwości zmiany kontekstu wykonania. Atutem zastosowania klastra będzie natomiast wciąż trwający proces replikowania sesji, co może okazać się nie bez znaczenia w przypadku awarii tego węzła.

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

org..server.SyncManager.

Serwer zarządca

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

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

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

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

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

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

Serwer ruchu

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

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

Dlatego w ramach pracy został napisany serwer ruchu jednocześnie działający jako serwer zarządca. Implementację

należy traktować jako wzorcowy przykład rozwiązania, a nie jako docelowy i w pełni działający serwer do przeki ero wy wania żądań.

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

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

Szczegóły implementacyjne

Dla każdego aktywnego serwera Tomcat trzymana jest pula połączeń (jej wielkość jest konfigurowalna). Każde połączenie jest zadaniem, które w momencie uruchomienia powoduje rozpoczęcie czytania z kanału komunikacyjnego. Jednocześnie obiekt zadania posiada metodę write (ByteBuffer , która umożliwia pisanie danych do połączenia. Przed ponownym uruchomieniem zadania (wywołaniem metody Restart o) ustawiane są referencje między zadaniem obsługującym połączenie z serwerem, a zadaniem obsługującym połączenie z klientem. W ten sposób później wszystkie odebrane informacje przez jeden obiekt zostają przesłane do drugiego (tworząc pomost dla danych). Różnice między zadaniami klienta a serwera występują

w zachowaniu na początku i końcu.

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

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

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

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

Wadą zaproponowanego rozwiązania jest brak interpretera nagłówka protokołu HTTP. Gdyby program był w stanie poprawnie

interpretować nagłówek mógłby zachowywać otwarte połączenia z każdym z węzłów, bez potrzeby ich odnawiania (do tego niezbędna byłaby kontrola długości przesyłanych danych). Wtedy również w przypadku trzymania połączeń typu Keep-Alive można by było dynamicznie zmieniać serwer, który obsłuży kolejne żądanie. Niestety stopień skomplikowania takiego interpretera wykracza poza ramy tej pracy.

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

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