STUDIA INFORMATICA 2013
Volume 34 Number 3 (113)

Marek WODA, Wojciech STOLARZ
Politechnika Wrocławska, Instytut Informatyki, Automatyki i Robotyki

ZWIĘKSZENIE EFEKTYWNOŚCI APLIKACJI TYPU SAAS POPRZEZ WYKORZYSTANIE MODELU ALOKACJI ZASOBÓW BAZOWANEGO NA DZIERŻAWIE ZASOBÓW NAJEMCOM

Streszczenie. Software-as-a-Service (SaaS) to paradygmat dystrybucji oprogramowania w chmurze. Reprezentuje on najwyższą warstwę oprogramowania w stosie chmury. Ponieważ większość dostawców usług w chmurze pobiera opłaty za korzystanie z jej zasobów, ważne jest tworzenie aplikacji efektywnie korzystających z jej zasobów. Jednym ze sposobów osiągnięcia optymalnego zużycia zasobów jest architektura *multi-tenant* aplikacji SaaS. Umożliwia on zastosowanie samodzielnego zarzadzania zasobami. W niniejszym artykule, w systemie SaaS z dzierżawą zasobów najemcom, przebadany został wpływ odpowiedniej alokacji zasobów na efektywność działania systemu. Model alokacji zasobów, uwzględniający najemców (*tenants*) jest jednym z podejść do optymalnego wykorzystania zasobów w chmurze. W porównaniu do tradycyjnego skalowania zasobów może wydatnie zmniejszyć koszty utrzymania aplikacji typu SaaS w środowisku chmury. Im więcej systemów SaaS z dzierżawą zasobów najemcom, tym więcej korzyści ten model może dostarczyć użytkownikom końcowym.

Słowa kluczowe: Model SaaS z dzierżawą zasobów najemcom, alokacja zasobów, efektywność

INCREASE EFFECTIVENESS OF A SAAS SYSTEM BY UTLIZATION OF TENANT-BASED RESOURCES ALLOCATION MODEL

Summary. Software-as-a-Service (SaaS) is a software distribution paradigm in cloud computing and represents the highest, software layer in the cloud stack. Since most cloud services providers charge for the resource use it is important to create resource efficient applications. One of the ways to achieve that is multi-tenant architecture of SaaS applications. It allows the application for efficient self-managing of the resources. In this paper the influence of tenant-based resource al-location model on cost-effectiveness of SaaS systems is investigated. The tenant-based resource allocation model is one of the methods to tackle under-optimal re-source utilization. When

compared to traditional resource scaling it can reduce the costs of running SaaS systems in cloud environments. The more tenant-oriented the SaaS systems are the more benefits that model can provide.

Keywords: tenant-based resource allocation, cost-effectiveness

1. Wstęp

Jednym z ostatnio omawianych rozwiązań problemów z nadmiernym wykorzystaniem/ niewykorzystaniem (*over-/ underutilization*) zasobów przez aplikacje typu *SaaS* w chmurze jest model alokacji zasobów bazowany na najemcach (*TBRAM – Tenant Based Resource Allocation Model*). Takie rozwiązanie zostało wprowadzone przez autorów [8] i przetestowane pod kątem efektywnego wykorzystania cykli procesora i zajętości pamięci. Autorzy wykazali przydatność modelu *TBRAM* przez redukcję czasu pracy serwera, wyrażoną w serwerogodzinach, jak również lepsze wykorzystanie zasobów. Autorzy przetestowali to podejście na zasobach prywatnej chmury, gdzie testowane były dwa tryby działania systemu (obciążenia narastające i szczytowe). Niestety środowisko autorskie mogło, co najwyżej, przybliżać realne warunki publicznej chmury, ale nią nie było. Celem niniejszego artykułu było dostarczenie odpowiedzi na pytanie, czy model *TBRAM* wart jest użycia w rzeczywistym środowisku chmury obliczeniowej, a nie tylko w izolowanym środowisku testowym. Przetestowanie autorskiego systemu *TBRAM* w publicznej, komercyjnej chmurze miało dostarczyć odpowiedzi na to pytanie. Dlatego głównym celem pracy była implementacja systemu *TBRAM* w myśl zaproponowanego opisu, jako przyszła część badawcza pracy [8].

Pomimo że w chmurze można automatycznie przydzielać i otrzymywać (na żądanie) zasoby, wciąż pojawiają się problemy z ich niedostatkiem, wynikające z nieadekwatnej do zapotrzebowania puli wspólnych zasobów. Często mamy do czynienia z dwoma przypadkami nieoptymalnego zużycia zasobów: ich nadmiernym wykorzystaniem lub niewykorzystaniem. Oba przypadki mają miejsce z powodu małej elastyczności obecnego modelu płatności (*pay--per-use*) za wykorzystanie zasobów chmury, stosowanego obecnie [17].

Z przewymiarowaniem systemu (*overprovisioning*) mamy do czynienia w momencie, gdy po przydzieleniu dodatkowych zasobów (w odpowiedzi na obciążenie szczytowe) nie są one zwalniane, mimo iż są już niepotrzebne, bo obciążenie spadło. W takim przypadku mamy do czynienia z niewykorzystaniem (i „marnowaniem") zasobów. Niedoszacowanie zasobów (*under provisioning*) przejawia się brakiem możliwości zapewnienia wymaganego poziomu usług z powodu niewystarczającej wydajności systemu (nasycenie zasobów). Taki stan systemu prowadzi do spadku jakości świadczonych usług i ostatecznie straty przychodów [3].

Płacenie za serwerogodziny stało się powszechne wśród komercyjnych operatorów chmur. Ma to też miejsce w przypadku *Elastic Cloud Computing* (*EC2*), za wykorzystanie zasobów którego firma *Amazon* pobiera opłaty – za każdą rozpoczętą godzinę pracy węzła *EC2*. Bardzo ważne jest, aby w pełni wykorzystać udostępnione zasoby, tak by zapłacić tylko za te zasoby, z których się rzeczywiście korzystało.

Jesteśmy wciąż w początkowej fazie rozwoju szeroko rozumianego *Cloud Computing*. Nikt jeszcze nie oczekuje działającego i sprawnego modelu płatności *pay-per-use* za aplikacje typu SaaS działające w chmurze, być może dlatego, że automatycznie skalowalna chmura nie będzie działać efektywnie w ten sposób [20]. Aby osiągnąć pożądaną skalowalność, należy projektować aplikację SaaS, mając to na uwadze, warto też odwołać się do [11]. Można skorzystać z architektury *multi-tenant*, by ułatwić zarządzanie zachowaniem aplikacji *SaaS*. Pozwala ona na wykorzystanie jednej instancji programu przez wielu użytkowników (najemców – *tenants*). Działaniem można ją porównać do klasy *singleton*, znanej z obiektowych języków programowania, zarządzającej tworzeniem i cyklem życia obiektów wywodzących się z tej klasy. Wsparcie wielu użytkowników jest również bardzo ważnym elementem przy projektowaniu aplikacji SaaS [5]. Można wyróżnić dwa podejścia: mnogość instancji (każdy najemca – użytkownik – ma swoją instancję aplikacji, działającą na wspólnych zasobach) oraz *native multi-tenancy* (pojedyncza instancja, działająca na rozproszonych zasobach) [2, 5]. Pierwsze podejście skaluje się z reguły dość dobrze dla niewielkiej liczby użytkowników, zaś dla większej (np. setki) wskazane jest, aby korzystać z drugiego.

1.1. Nadmierne wykorzystanie zasobów (Overutilization)

Termin „punkt wyczerpania” zasobów (*point of exhaustion*) jest często używany w odniesieniu do nadmiernego wykorzystania zasobów. Jest to sytuacja, w której co najmniej jeden z zasobów jest w pełni wykorzystany, na przykład, gdy nastąpi 100% wykorzystanie procesora lub pamięci [16]. Definicja ta wydaje się być zasadna w wielu prostych przypadkach. Jednakże, w przypadku chmury, jest zbytnio uproszczona. Autorzy [12] proponują inną definicję, według której punkt wyczerpania jest maksymalnym obciążeniem, które może być przypisane pojedynczej maszynie wirtualnej bez zmniejszania, w tym samym czasie, jej wydajności. Działanie powyżej „punktu wyczerpania (zasobów)” definiuje nasycenie maszyny. Ta „nowa” definicja punktu wyczerpania wymaga pomiaru przepustowości maszyny wirtualnej wraz z wykorzystaniem jej zasobów (np. CPU lub pamięci). Autorski, testowy system SaaS wykorzystuje narzędzie *JMeter* w połączeniu z metrykami narzędzia *CloudWatch*, w celu wychwycenia tego momentu. Aplikacja SaaS jest obciążana żądaniami HTTP generowanymi przez narzędzie. Następnie obliczana jest przepustowość systemu przez podzielenie liczby

żądań HTTP przez czas od początku pierwszego do końca trwania ostatniego żądania. Dzięki temu możliwy jest pomiar czasu przetwarzania między żądaniami.

W pracach [12, 13, 15, 16, 18, 19] autorzy koncentrują się na wydajności systemu do momentu wykrycia punktu nasycenia (przegięcia). Ilekroć przepustowość spada, a wykorzystanie zasobów maszyn wirtualnych rośnie – występuje punkt przegięcia. W artykule przyjęto taką samą konwencję.

Autorski system SaaS używa wspomnianych punktów przegięcia, aby wykryć stan nadmiernego wykorzystanie zasobów (*overutilization*) na danej maszynie wirtualnej. Wszystkie maszyny wirtualne (wraz z serwerem *Tomcat*) są monitorowane (zbierane są dane o wykorzystaniu zasobów i monitorowana jest ich wydajność). Brane są pod uwagę podstawowe metryki: wykorzystania procesora i zużycia pamięci sterty przez wirtualną maszynę Javy. Na podstawie tychże pomiarów można oszacować czy maszyna wirtualna (*VM*) jest nasycona czy nie.

Ogólnie rzecz biorąc, gdy maszyna wirtualna (*VM*) jest nasycona, procesy systemu operacyjnego zaczynają konsumować coraz więcej zasobów wykonując przy tym coraz wolniej procesy użytkownika. Ma to negatywny wpływ na czas reakcji systemu, a zatem na samego użytkownika. Należy dążyć do uniknięcia tego zjawiska, mimo iż nie ma ono bezpośredniego wpływu na koszt (nie płacimy dodatkowo za wysokie obciążanie maszyn wirtualnych), ale dlatego, że może ono prowadzić do zmniejszenia wpływów za świadczenie usług z powodu odpływu niezadowolonych użytkowników.

1.2. Niewykorzystanie dostępnych zasobów (*Underutilization*)

Z ekonomicznego punktu widzenia niewykorzystanie dostępnych zasobów to czysta strata pieniędzy, płacimy za coś, czego nie potrzebujemy lub nie używamy. Z punktu widzenia użytkownika końcowego taka sytuacja jest niezauważalna, natomiast z punktu widzenia operatora już tak, opłaty za niewykorzystane zasoby są po prostu marnowaniem pieniędzy. Bardziej formalnie [4, 7] *underutilization* opisuje sytuację, gdy część zaalokowanych zasobów chmury nie jest wykorzystywana przez działające maszyny wirtualne. Oczywiście, jest prawie niemożliwe, aby zapewnić 100% wykorzystanie zasobów przez cały czas działania systemu. Zatem nieuniknione jest, by niewielki procent zasobów pozostawał niewykorzystany. Niewykorzystanie zasobów w chmurze można wyrazić przez ilość zasobów dostępnych do wykorzystania. Zgodnie z [4] dostępny zasób jest marnowany, kiedy przydzielany jest maszynie VM, nie powodując jego skonsumowania, np. obciążenie jest alokowane na dwu maszynach wirtualnych, a z powodzeniem mogłoby być obsłużone przez jedną (bez nasycenia), zatem zasoby na obu maszynach są marnowane. W celu sprawdzenia, czy obciążenie z jednej maszyny może być zaalokowane na innej, należy obliczyć możliwe kombinacje obciążeń

maszyn wirtualnych wraz ze zużyciem wybranych zasobów. Jednym ze sposobów rozwiązania tego problemu jest zastosowanie algorytmu plecakowego, co zaproponowano w artykule bazowym [8]. Ilość wykorzystywanych zasobów przypisano do wag elementów w plecaku. Wartością pojedynczego elementu z plecaka określono dostępną ilością zasobu w maszynach wirtualnych. W przypadku pamięci sterty Javy, za tę wartość przyjmuje się ilość pamięci, która jest nadal dostępna. Pojemność plecaka jest określana przez maksymalną ilość dostępnego (dla maszyny wirtualnej) zasobu, któremu staramy się przypisać obciążenie. Dzięki zastosowaniu tej metody, otrzymujemy maksymalną liczbę maszyn wirtualnych, która może być użyta. Ta liczba była zastosowana do pomiaru poziomu niewykorzystania zasobów (*underutilization)*. Im mniejsza liczba, tym lepsze wykorzystanie zasobów.

1.3. Szacowanie kosztów

Uruchomienie systemu w chmurze publicznej daje nam jeszcze jeden sposób, aby ocenić zasadność takiego podejścia. Prawie każda operacja wykonana w chmurze jest rejestrowana i dodana do naszego rachunku. Płacimy za wysłane żądania, przestrzeń dyskową, serwerogodziny pracy maszyn wirtualnych i wiele innych rzeczy. Rachunek wystawiony za te usługi daje prawdopodobnie najdokładniejsze oszacowanie efektywności finansowej porównywanych systemów. Dzięki niemu wiemy, jaką cenę trzeba zapłacić za wykorzystanie aplikacji w chmurze. Podczas badań usługa *CloudWatch* zbierała dane dotyczące wykorzystania zasobów środowiska chmury AMAZON. Oba systemy SaaS (system bazowy [8] i autorski TBRAM) były objęte tym samym zestawem testów, liczba żądań obsługi była dokładnie taka sama. Główna różnica między nimi była w liczbie używanych maszyn wirtualnych. Różnica ta znalazła odzwierciedlenie w rachunku finansowym. Porównanie kosztów uruchamiania aplikacji SaaS w obu systemach pokazała zauważalną różnicę na korzyść autorskiej implementacji modelu TBRAM w stosunku do tradycyjnego sposobu skalowania zasobów.

2. Prace powiązane

Autorzy w pracy [5] proponują profile, jako podejście do skalowania w chmurze. Starają się wykorzystywać najlepsze praktyki i wiedzę w celu tworzenia odpowiednich, skalowalnych profili. Profil zawiera informacje, które pomagają scharakteryzować serwer w zakresie jego możliwości. Kiedy skalowanie jest uruchomione, odpowiedni profil brany jest pod uwagę. W [9] autorzy proponują zestaw narzędzi wykorzystujący mechanizmy języka Java, aby wesprzeć obsługę podejścia typu *multi-tenacy*. Stosują elementy kontekstu, by śledzić aplikacje uruchomione przez Java Virtual Machine. To z kolei pozwala na odróżnianie każdego

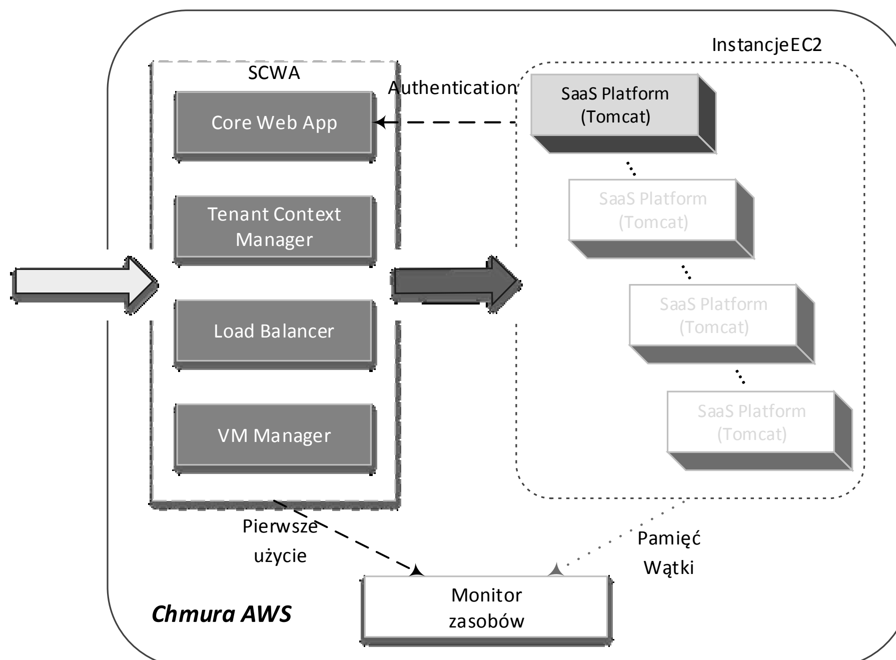
najemcy (*tenant*). Ta wiedza jest później wykorzystana w celu oszacowania użycia zasobów przez każdego z najemców i do wyliczania opłaty za wykorzystanie mocy obliczeniowej. W pracy [6] autorzy stawiają zarówno na inteligentne mapowanie zasobów, jak i efektywne zarządzanie maszynami wirtualnymi, które w znacznym stopniu wpływają na koszty działania aplikacji w chmurze. W pracy [10] opisane są trzy główne elementy, które wpływają na wydajność maszyny wirtualnej; są to: pomiar jej parametrów, modelowanie i zarządzania zasobami. Autorzy opisują tam model oszacowania potencjalnej straty wydajności, przy jednoczesnej konsolidacji maszyn wirtualnych. Amazon wykorzystuje własne narzędzie automatycznego skalowania [1] do zarządzania maszynami wirtualnymi poprzez predefiniowane (lub definiowane przez użytkownika) *triggery*. Jest to mechanizm specyficzny dla platformy Amazon EC2. W [8] autorzy implementują mechanizm alokacji zasobów bazowany na dzierżawie zasobów najemcom w prywatnej chmurze Eucalyptus, w której zostały przeprowadzone testy symulacyjne z narastającymi i szczytowymi obciążeniami. Zaprezentowane wyniki wykazały znaczącą redukcję serwerogodzin w porównaniu do tradycyjnego modelu skalowania zasobów. Model opierający się na dzierżawie najemców znacząco poprawił wykorzystanie zasobów chmury testowej w aplikacji typu SaaS. Co więcej, zostały tam zaproponowane formalne metody szacowania nadmiernego wykorzystania/niewykorzystania zasobów wirtualnych. Pomiary (cykli CPU i zajętości pamięci) zaprojektowane zostały specjalnie na potrzeby aplikacji SaaS. W tymże artykule [8] zaprezentowany został efektywny kosztowo model alokacji zasobów, bazowany na dzierżawie zasobów najemcom dla aplikacji działającej w modelu SaaS. W tym artykule będzie on nazywany systemem bazowym.

3. Opis systemu

System *TBRAM* wykorzystuje trzy techniki, wspierające podejście *multi-tenancy*. Pierwszą z nich jest izolacja najemców – separacja ich kontekstu. Została ona zrealizowana przez uwierzytelnianie najemców (*tenant-based authentication*) i trwałe struktury danych (*data persistence*) – stanowi część platformy SaaS (poprzez instancje Tomcat). Drugą jest wykorzystanie alokacji maszyn wirtualnych, bazowanej na najemcach. Dzięki niej można obliczyć rzeczywistą liczbę maszyn wirtualnych wymaganych do obsługi przez każdego klienta (*tenant*) w danej chwili. Trzecia, realizuje równoważenie obciążenia bazowane na liczbie najemców[1] (*tenant-based load balancing*). Pozwala ona na dystrybucję obciążenia wewnątrz maszyn wirtualnych, w zależności od zapotrzebowania na zasoby poszczególnych klientów (*tenants*). Schemat architektury tego systemu został przedstawiony na rys. 1. Linią przerywa-

[1] Technika będzie szerzej opisana w osobnym artykule.

ną oznaczono komunikację do *Web Services*. Warto zauważyć, że element *SCWA* (*SaaS Core Web App*) (na rysunku) był jedynym, który różni autorski system od systemu bazowego [10]. System spełnia postulaty modelu TBRAM.



Rys. 1. Architektura systemu TBRAM
Fig. 1. TBRAM system architecture

3.1. Izolacja najemców (*Tenant-based isolation*)

Jest oczywiste, że sytuacja, gdy jeden najemca może uzyskać dostęp (i wpływać) na nie swoje dane jest nie do zaakceptowania w jakimkolwiek (nawet nie komercyjnym) rozwiązaniu, zatem aby system działał prawidłowo, wymagana jest izolacja każdego najemcy (*tenant*). Model TBRAM zaleca izolację na niskim poziomie, gdyż znacząco poprawia ona skalowalność [2]. Izolacja najemców w podejściu TBRAM może być implementowana na dwa sposoby: poprzez trwałe struktury danych (*data persistance*) i przez mechanizmy uwierzytelniania najemców (*tenant-based authentication*). Warto nadmienić, że izolacja najemców została również zastosowana w systemie bazowym (odniesienia), a oba systemy, autorski i bazowy, korzystają z bazy najemców (*multi-tenant*), co ma wpływ jedynie na platformę SaaS, a nie na koncepcję *SCWA*. Dzięki temu w obu przypadkach dwie różne aplikacje SaaS wykorzystywały taką samą platformę, przez co zminimalizowany został potencjalny negatywny wpływ różnic (platform) na wyniki badań.

W warstwie trwałości danych (*persistence layer*) zaproponowano współdzieloną bazę danych (*Shared Database* / *Shared Schema*), ponieważ miała ona mały wpływ na wykorzystanie sprzętu i pozwalała na utworzenie największej liczby najemców per serwer [14]. Aby logicznie odseparować dane, w każdej tabeli bazy zostało wprowadzone pole *TenantID*. Została

użyta biblioteka JoSQL, która umożliwiła wykonywanie zapytań SQL na kolekcjach obiektów języka Java. Biblioteka ta była używana przez *Struts2 interceptors*, aby wstępnie przetwarzać dane najemców. Dodatkowo, adnotacje (*annotations*) języka Java używane były do oznaczania miejsc w kodzie, który był właściwy dla zachowań najemców (*tenant-based*). To podejście wydawało się najskuteczniejszym sposobem na wdrożenie podejścia *mutli-tenancy* (separacja danych pobieranych przez wstępne ich filtrowanie).

Wykorzystanie *przeplatania*, jako realizacji postulatu programowania aspektowego (*aspect-oriented programming*) ma wiele zalet. Najważniejszą jest to, że z jednego miejsca w kodzie można mieć wpływ na dowolną klasę, oznaczoną przez adnotację. Dodatkowo, jedyną rzeczą wymagająca zmiany w istniejącym kodzie aplikacji, aby umożliwić wykorzystanie pełnego podejścia *mutli-tenancy*, jest wprowadzenie adnotacji. Prawdopodobnie jest to najbardziej pospolity sposób, by rozdzielić najemców w istniejącej aplikacji.

Uwierzytelnianie najemców (*Tenant-based authentication*) było drugą techniką w celu osiągniecia pełnej izolacji klientów. W myśl założeń modelu TBRAM należy wdrożyć ją w *SCWA (SaaS Core Web App)*. Podczas uwierzytelniania każdy użytkownik wiązany jest unikalnym identyfikatorem (*Tenant ID*).

Od tego momentu może on uzyskać dostęp tylko do tych danych, do których jest uprawniony (bez uwierzytelnienia nikt nie ma dostępu do danych). W systemie TBRAM sugerowane jest też użycie list kontroli dostępu (*ACL Access Control Lists*), które zdecydowano się pominąć, ze względu na dodatkowe utrudnienia w implementacji, a ich zastosowanie i tak nie ma wpływu na wyniki badań. Dla uproszczenia postanowiono dać pełny dostęp do wszystkich aplikacji SaaS wszystkim użytkownikom. Było to konieczne, aby zebrać informacje o najemcach (*tenant data*) z każdego punktu w systemie. Model TBRAM sugeruje również użycie *cookie*s i kontekstu servletu. Natomiast autorzy [8] w swoim rozwiązaniu do uwierzytelniania najemców wykorzystali lokalny klaster *Tomcat*. W autorskim podejściu zdecydowano się nie używać instancji Tomcata uruchomionych w trybie klastra, ze względu na nadmierny koszt wymiany informacji między wszystkimi węzłami. Podczas działania w różnych (pod-)sieciach, może to mieć negatywny wpływ na wydajność systemu. Serwer *Tomcat* w wersji 7 obsługuje replikację sesji w trybie *all-to-all*, ale zamiast tego mechanizmu użyty został dedykowany (niezależny od platformy) *web service*, aby dostarczać informacje o najemcach. Każdy użytkownik przy próbie pierwszego dostępu do dedykowanej mu maszyny wirtualnej był poddawany kontroli w bloku *SCWA* i w tym miejscu podejmowana była decyzja o jego uwierzytelnieniu i autoryzacji. Jeśli uwierzytelnienie i autoryzacja przebiegły pomyślnie, dane o tym fakcie zapisywane były lokalnie (w kontekście sesji), by przy następnym dostępie do VM użytkownik nie był dodatkowo weryfikowany przez scentralizowany *web service SCWA*. Dane o autoryzacji otrzymują tylko te maszyny wirtualne, które tego wymagają w danym momencie.

Model TBRAM definiuje obiekt *Tenant Context*, który zawiera informacje o: unikalnym identyfikatorze najemcy *TenantID*, aktywnych użytkownikach i przypisanym im maszynom wirtualnym itp. *Tenant Context Manager* jest używany do zarządzania obiektami typu *Tenant Context*. Dzięki niemu informacja o stanie najemcy jest dostępna dla wszystkich innych usług. *Tenant Context* pozwala na izolowanie każdego wysłanego żądania (do platformy) na podstawie danych użytkownika najemcy. Pomimo iż użytkownicy fizycznie współdzielą tę samą aplikację SaaS (i warstwę trwałości danych), nadal pozostają logicznie izolowani przez ich *Tenant Context*. Obiekty *Tenant Context* pomagają stworzyć natywne podejście *multi-tenancy* i dzięki nim użytkownicy nie mają pojęcia, że dzielą te same zasoby.

3.2. Przypisywanie maszyn wirtualnych najemcom (*Tenant-based VM allocation*)

Technika przypisywania maszyn wirtualnych najemcom wykorzystywana jest do określenia liczby maszyn, potrzebnych danemu najemcy do efektywnego świadczenia usług. Wykorzystano tu pomysł profili w połączeniu z monitorowaniem usług realizowanych w aplikacji SaaS. Profil używany w środowisku testowym (z artykułu bazowego) był odpowiednikiem profilu m1.small EC2 w chmurze Amazon, a jego parametry były następujące: 1 rdzeń procesora, 1 GB pamięci, do 800 MB pamięci sterty JVM, 100 to maksymalna liczba użytkowników, których miała obsłużyć maszyna wirtualna ($Threads_{max} = 200$). Na potrzeby badań został użyty analogiczny profil, by aplikacja SaaS wykorzystywała profil m1.small. Główną różnicą była maksymalna liczba użytkowników – ograniczona w tym przypadku do 50. Dane z profilu, wraz aktualnymi wartościami metryk były brane pod uwagę do ustalania liczby instancji maszyn wirtualnych. *Tenant Context Manager* był odpowiedzialny za wyznaczenie wag każdemu obiektowi *Tenant Context*, a wagi te były później wykorzystywane do wyznaczania liczby maszyn wirtualnych.

Wagi definiuje następujący wzór:

$$TenantContextWeight = users_{active} \times \left(\frac{heapSize}{Threads_{\max}} \right),$$

$$TenantContextWeight = users_{active} * \left(\frac{heapSize}{Threads_{max}} \right) \quad (1)$$

gdzie: $users_{active}$ to użytkownicy, których sesja nie wygasła, *heapSize* jest ilością pamięci przypisanej do JVM (ustawianej w profilu), a $Threads_{max}$ $Threads_{max}$ to dozwolona liczba równoległych wątków dla platformy SaaS. Fragment w nawiasie może być traktowany jako uśrednienie wykorzystania pamięci na wątek dla danego profilu. Powyższy wzór jest oszacowaniem wymaganej pamięci dla danej liczby aktywnych użytkowników. Poniższy wzór służy do obliczenia pojemności maszyny wirtualnej:

$$capacity_{VM} = heapSize - \left(\left(\frac{heapSize}{Threads_{\max}} \right) \times Threads_{platform} \right)$$

$$TenantContextWeight = users_{active} * \left(\frac{heapSize}{Threads_{max}} \right) \quad (2)$$

W powyższym wzorze odejmowane jest aktualne wykorzystanie pamięci z maksymalnej dozwolonej ilości, opisanej w profilu. Bieżące zużycie pamięci wyliczane jest przez pomnożenie liczby wątków (w stanie gotowości) przez średnie zużycie pamięci na wątek.

Z tego wzoru wiemy, ile pamięci jest dostępne wyłącznie dla użytkowników danej instancji aplikacji SaaS. Warto zwrócić uwagę, że część pamięci przypisanej do JVM jest zużywana przez wątki serwera *TOMCAT* oraz wątki platformy tylko do uruchomienia usługi. Wszystkie wątki, które zostały utworzone, są używane na potrzeby danego użytkownika. Mając to na uwadze, jesteśmy w stanie oszacować początkowe (rzeczywiste) zużycie zasobów.

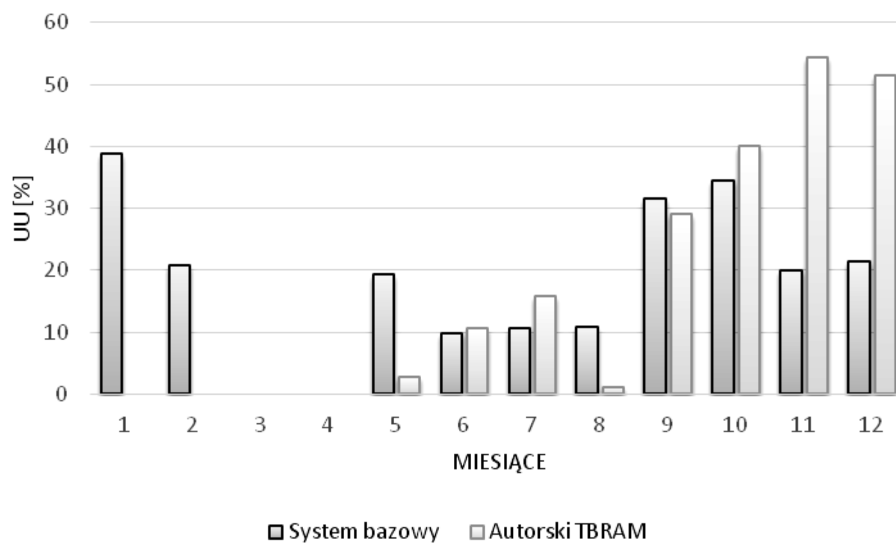
Podejście TBRAM sugeruje użycie algorytmu plecakowego do obliczanie minimalnej liczby instancji, wymaganych do pokrycia bieżącego zapotrzebowania na obciążenie. Wartość zwracana przez algorytm jest jedyną, ponieważ nie jest brane pod uwagę bieżące przypisanie najemców do obecnie dostępnych maszyn wirtualnych. W algorytmie użyto wartości wyliczanych za pomocą wzorów (1) i (2). Do szybkiego rozwiązania „problemu plecakowego" wykorzystano dynamiczny sposób programowania, który został skonceptualizowany w postaci narzędzia *Tenant-Based VM Calculator*. Wyniki jego pracy określają liczbę instancji maszyn wirtualnych, generowanych przez *VM Managera* z chmury AWS. Podstawowym, ale nie jedynym źródłem informacji o wymaganej liczbie instancji jest algorytm plecakowy. Mając na uwadze, że często zdarza się, że najbardziej zaawansowane estymacje zawodzą, co prowadzi do rozbieżności między rzeczywistością a zakładanym stanem aplikacji, zdecydowano się dodatkowo wprowadzić również „czynnik ludzki", a mianowicie badać kolejne żądania obsługi i jeśli one są nieobsługiwane, to automatycznie tworzona jest nowa instancja maszyny wirtualnej (w imieniu użytkownika wysyłane jest żądanie do *VM Managera*, który ją tworzy).

4. Wstępne wyniki badań

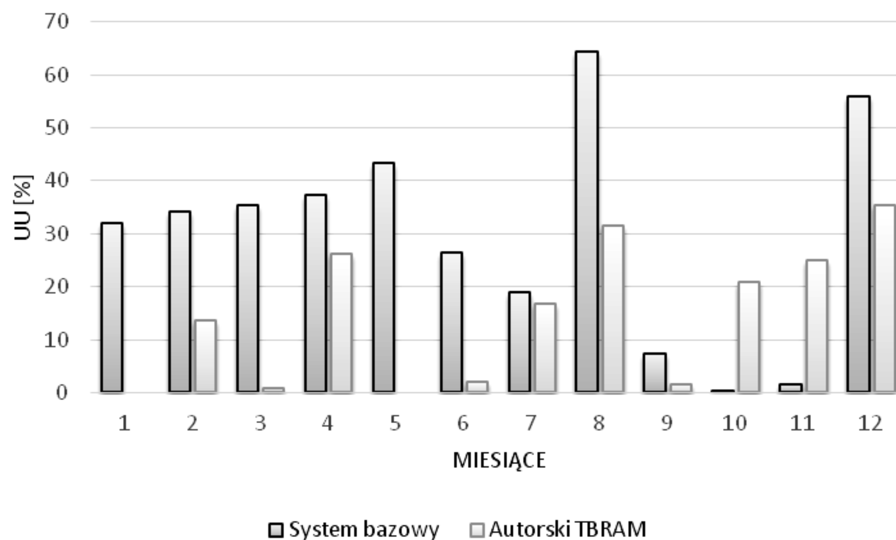
Rozdział ten zawiera wyniki uzyskane po przeprowadzonych testach wstępnych systemu autorskiego. Zawarte są w nim dane dotyczące czasu pracy (serwerogodzin), nadmiernego wykorzystania/niewykorzystania zasobów oraz generowanych kosztów działania systemów. Usługa *Amazon CloudWatch* posłużyła do zbierania danych pomiarowych podczas trwania testów obciążeniowych. Narzędzie to pozwala na przeglądanie kilku podstawowych statystyk

danych w postaci wykresów. Jednakże, w celu wykonania bardziej zaawansowanej analizy zostały wykorzystane jego surowe dane.

Na wykresie (rys. 2) możemy zobaczyć porównanie systemów bazowego i autorskiego TBRAM pod względem niewykorzystania zasobów (*underutilization*), przy obciążeniu narastającym. Warto zauważyć, że w ciągu pierwszych czterech miesięcy, w symulowanym roku pracy, problem niewykorzystania zasobów nie istniał w przypadku systemu TBRAM. W połowie roku oba systemy były porównywalne, a pod koniec system bazowy stał się znacznie bardziej skuteczny. Ogólnie można stwierdzić, że w przypadku testów narastającego obciążenia oba systemy są porównywalne pod kątem niewykorzystania zasobów.



Rys. 2. Niewykorzystanie zasobów w systemach testowych – obciążenie narastające
Fig. 2. Resource underutilization during the incremental workload tests of the systems



Rys. 3. Niewykorzystanie zasobów w systemach testowych – obciążenie szczytowe
Fig. 3. Resource underutilization during the peak – based workload tests of the systems

Na wykresie (rys. 3) przedstawiono wyniki pomiaru niewykorzystania dostępnych zasobów (parametr *Underutilization*) przy obciążeniu szczytowym systemów. Zauważalnie lepsze są wyniki autorskiego systemu TBRAM. Dla większości roku, autorski system TBRAM był dużo wydajniejszy niż system bazowy (w zakresie sumarycznego wykorzystania zasobów). W wielu przypadkach autorski system był lepszy o 50% i więcej. Dynamiczne alokowanie maszyn wirtualnych (w autorskim systemie TBRAM) wykazywało swą wyższość nad podejściem zastosowanym w systemie bazowym, gdy obciążenie systemów gwałtownie się zmieniało. Warto zwrócić uwagę, że średnie wartości UU% obu systemów są niższe niż w przypadku tradycyjnego systemu alokowania zasobów. Wykorzystano test t-studenta w celu potwierdzenia, że wyniki uśrednień (%UU) są statystycznie różne. Chciano tym potwierdzić, że autorski system TBRAM był znacząco lepszy od bazowego.

Poza oceną techniczną obu systemów, wykonano analizę opłacalności finansowej zastosowania tychże systemów. Jako dane do analizy posłużyły: rachunek za usługi firmy Amazon oraz narzędzie *Simple AWS Monthly Calculator*. Skupiono się na różnicach w koszcie pomiędzy dwoma systemami, a nie całościowym koszcie przedsięwzięcia czy jego sensowności. Dlatego w porównaniu kosztów zostały uwzględnione tylko te części systemów, które były różne.

Pod uwagę brane były tylko maszyny wirtualne platformy SaaS oraz system równoważenia obciążenia (*load balancer*). Wszystkie inne elementy środowiska testowego, takie jak bazy danych, RCM, klaster JMeter nie były brane pod uwagę przy analizie finansowej. Oba środowiska testowe były poddane jednakowym próbom, działały pod tym samym obciążeniem, a wykorzystanie baz danych i innych zasobów było dokładnie takie samo. Z tego względu podjęto decyzję o usunięciu elementów wspólnych z porównania, aby zwiększyło to klarowność analizy. Dodatkowo, w porównaniu zostały ujęte koszty instancji EC2 platformy SaaS i jej systemów równoważenia obciążenia. Do obliczeń przyjęto ceny usług *Amazon EC2* dla UE z 2012 roku, które zostały ujęte w tabeli 1.

Tabela 1

Koszt każdej rozpoczętej serwerogodziny pracy instancji EC2

	Typ instacji EC2		
Koszt	m1.small	m1.medium	ELB
serwerogodziny	0,085 USD	0,170 USD	0,028 USD

W tabeli 1 przedstawiono koszt każdej rozpoczętej serwerogodziny pracy instancji EC2 w zależności od jej typu. Środowisko testowe wykorzystywało instancje EC2 m1.small. Element SCWA systemu TBRAM była umieszczony w instancji m1.medium, tak jak to miało miejsce w systemie bazowym.

Tabela 2

Koszt GB danych przesyłanych przez chmurę AWS

	Typ transferu danych

Koszt	Transfer out	ELB in/out	ELB data proc.
per GB	0,120 USD	0,010 USD	0,008 USD

W tabeli 2 przedstawiono z kolei koszt GB danych przesyłanych przez chmurę AWS. Warto podkreślić, że większość usługodawców pobiera opłaty za ruch w jedną stronę (ruch wychodzący), natomiast w przypadku ELB płaci się za ruch w obu kierunkach, jak również za każdy GB danych przetworzonych w chmurze.

Tabela 3

Koszty wygenerowane przez poszczególne części obu systemów

	Narastające	Szczytowe	Suma
System bazowy	13,40 USD	21,74 USD	35,14 USD
Autorski TBRAM	11,82 USD	17,85 USD	29,67 USD
Razem	25,22 USD	39,59 USD	64,80 USD

W tabeli 3 przedstawiono koszty wygenerowane przez poszczególne części obu systemów. Łatwo można dostrzec, że dla obu typów obciążeń, system TBRAM sprawdza się lepiej (jest bardziej ekonomiczny) – pozwala na redukcję kosztów średnio o 15,58%.

5. Podsumowanie

W artykule przenalizowano dwa aspekty efektywności aplikacji SaaS. Pierwszy brał pod uwagę parametry techniczne (obciążeniowe) systemów (wyrażone m.in. przez liczbę serwerogodzin – do przedstawienia czasu pracy podczas testów), drugi analizował systemy pod kątem opłacalności i opierał się na rachunku rozliczeniowym dostarczonym przez *AWS Amazon* za wykorzystanie zasobów chmury. Autorski system TBRAM zużył blisko 20% mniej serwerogodzin w przypadku przyrostowego testu obciążenia i ponad 30% mniej serwerogodzin w przypadku testów przy obciążeniu szczytowym. System ten wygenerował również blisko 16% mniejszy koszt niż system bazowy. Autorzy sprawdzili również, czy model TBRAM wpływa na poprawę wydajności aplikacji SaaS. Badanie wykazały, że model ten statystycznie znacząco (z 97,5% dokładnością) poprawił wykorzystanie zasobów w przypadku obciążenia szczytowego w stosunku do systemu bazowego. Niestety, implementacja aplikacji zgodnie z modelem TBRAM wprowadza pewne trudności implementacyjne i wydłuża czas trwania projektu. Biorąc pod uwagę uzyskane wyniki można stwierdzić, że alokacja zasobów oparta na dzierżawie zasobów najemcom ma duży wpływ na opłacalność (zwłaszcza finansową) aplikacji typu SaaS.

W dalszych badaniach autorzy skupią się na wprowadzeniu własnej metody równoważenia obciążenia na potrzeby architektury TBRAM, tak by porównać wyniki działania systemu opartego na ELB w zakresie skalowania zasobów dzierżawionym najemcom. Oczekuje się,

że dynamiczne skalowanie zasobów na podstawie autorskiego podejścia istotnie zmniejszy liczbę serwerogodzin.

BIBLIOGRAFIA

1. Amazon Auto Scaling. http://aws.amazon.com/autoscaling/. Dostęp 2012-12-07.
2. Architecture Strategies for Catching the Long Tail: 2006. http://msdn.microsoft.com/ /en-us/library/aa479069.aspx. Dostęp 2012-12-07.
3. Armbrust M. et al.: Above the Clouds: A Berkeley View of Cloud Computing. Technical Report #UCB/EECS-2009-28. Electrical Engineering and Computer Sciences University of California at Berkeley, 2009.
4. Bientinesi P. et al.: HPC on Competitive Cloud Resources. Handbook of Cloud Computing. B. Furht and A. Escalante, eds. Springer US, 2010, p. 493-516.
5. Chang J. G. et al.: A framework for native multi-tenancy application development and management. 2007 9th IEEE International Conference on e-Commerce Technology and the 4th IEEE International Conference on Enterprise Computing, e- Commerce, and e-Services, 23-26 July 2007 (Piscataway, NJ, USA, 2007), p. 470-477.
6. Chen Y. et al.: An Efficient Resource Management System for On-Line Virtual Cluster Provision. IEEE International Conference on Cloud Computing, 2009. CLOUD '09 (Sep. 2009), 2009, p. 72-79.
7. Dyachuk D., Deters R.: A solution to resource underutilization for web services hosted in the cloud. Confederated International Conferences on On the Move to Meaningful Internet Systems, OTM 2009: CoopIS 2009, (Vilamoura, Portugal, 2009), p. 567-584.
8. Espadas J. et al.: A tenant-based resource allocation model for scaling Software-as-a-Service applications over cloud computing infrastructures, 2011.
9. Hong C. et al.:An end-to-end methodology and toolkit for fine granularity SaaS-ization. 2009 IEEE International Conference on Cloud Computing (CLOUD), 21-25 Sept. 2009 (Piscataway, NJ, USA, 2009), p. 101-108.
10. Iyer R. et al.: VM3: Measuring, modeling and managing VM shared resources. Computer Networks. 53, 17 (Dec. 2009), p. 2873-2887.
11. Mc Evoy G.V., Schulze B. Using clouds to address grid limitations. 6th International Workshop on Middleware for Grid Computing, MGC'08, held at the ACM/IFIP/ /USENIX 9th International Middleware Conference, (Leuven, Belgium 2008).

12. Meng X. et al.: Efficient resource provisioning in compute clouds via VM multiplexing. 7th IEEE/ACM International Conference on Autonomic Computing and Communications, ICAC-2010 (Washington, DC, United states, 2010), p. 11-20.
13. Mishra A.K. et al.: Towards Characterizing Cloud Backend Workloads: Insights from Google Compute Clusters. Performance Evaluation Review. 37, 4 (Mar. 2010), p. 34-41.
14. Multi-Tenant Data Architecture: 2006. http://msdn.microsoft.com/en-us/library/ /aa479086.aspx. Dostęp: 2012-09-07.
15. Paroux G. et al.: A Java CPU calibration tool for load balancing in distributed applications. Proceedings – ISPDC 2004: Third International Symposium on Parallel and Distributed Computing/HeteroPar '04: Third International Workshop on Algorithms, Models and Tools for Parallel Computing on Heterogeneous Networks, (Cork, Ireland 2004), 155–159, 2004.
16. SaaS Capacity Planning: Transaction Cost Analysis Revisited: 2008. http://msdn.microsoft.com/en-us/library/cc261632.aspx. Dostęp: 2012-09-07.
17. Stillwell M. et al.: Resource allocation algorithms for virtualized service hosting platforms. Journal of Parallel and Distributed Computing. 70, 9 (2010), p. 962-974.
18. Wee S., Liu H.: Client-side load balancer using cloud. 25th Annual ACM Sympo-sium on Applied Computing, SAC 2010, (Sierre, Switzerland 2010), p. 399-405.
19. Wu Q., Wang Y.: Performance testing and optimization of J2EE-based web applications. 2nd International Workshop on Education Technology and Computer Science, ETCS 2010, March 6, 2010 – March 7, 2010 (Wuhan, Hubei, China 2010), p. 681-683.
20. Yang J. et al.: A profile-based approach to just-in-time scalability for cloud applications. CLOUD 2009 – 2009 IEEE International Conference on Cloud Computing, September 21, 2009- September 25, 2009 (Bangalore, India 2009), p. 9-16.

Wpłynęło do Redakcji 16 marca 2013 r.

Abstract

Cloud computing is getting more and more interest with every year. It is an approach that allows Internet based applications to work in distributed and virtualized cloud environment. It is characterized by on-demand resources and pay-per-use pricing. Software-as-a-Service (SaaS) is a software distribution paradigm in cloud computing and represents the highest software layer in the cloud stack. Since most cloud services providers charge for the resource

use it is important to create resource efficient applications. One of the way to achieve that is multi-tenant architecture of SaaS applications. It allows the application for efficient self-managing of the resources Objectives. In this study the influence of tenant-based allocation model on cost-effectiveness of SaaS systems is investigated. Authors try to find out weather that model can decrease the system's actual costs in commercial public cloud environment.

There are two authorial SaaS systems implemented: first tenant-unaware and then using tenant-based allocation model. They are deployed on Amazon public cloud environment. Tests focused on measuring over- and underutilization are conducted in order to compare cost-effectiveness of the solutions. Public cloud provider's bill service is used as a final cost measure. Several strategic release planning models are found and mapped in relation to each other, and a taxonomy of requirements selection factors is constructed. Similar research was done, but proposed system is deployed on commercial public cloud instead of previously tested private one. There are many models are related to each other and use similar techniques to address the release planning problem. We conclude that several requirement selection factors are covered in the different models, but that many methods fail to address factors such as stakeholder value or internal value. Moreover, there is a need for further empirical validation of the models in full scale industry trials.

Adresy

Marek WODA: Politechnika Wrocławska, Instytut Informatyki, Automatyki i Robotyki, ul. Janiszewskiego 11/17, Wrocław, Polska, marek.woda@pwr.wroc.pl

Wojciech STOLARZ: Politechnika Wrocławska, Instytut Informatyki, Automatyki i Robotyki, ul. Janiszewskiego 11/17, Wrocław, Polska, voytec0dh@gmail.com