Jarosław Błąd, e-point SA

Problemy w dużych aplikacjach internetowych J2EE

Problemy w dużych aplikacjach internetowych Java 2 Enterprise Edition (J2EE) możemy zasadniczo podzielić na dwie kategorie. Pierwszą stanowią problemy związane z implementacją systemów, drugą – problemy występujące w trakcie eksploatacji i utrzymania systemów. Ponieważ na temat implementacji dużych systemów wiele już napisano i powiedziano, w tym artykule skupię się wyłącznie na drugiej grupie problemów, którym w literaturze poświęca się zdecydowanie mniej uwagi. Swoje rozważania będę opierać na doświadczeniach firmy e-point SA w rozwiązywaniu problemów utrzymaniowych i eksploatacyjnych w dużych aplikacjach internetowych.

Na początek spróbujmy zdefiniować, **czym jest „duży system”**.

Podstawowym kryterium jest **wolumen użytkowników**, **transakcji** i **danych**. Jeśli jeden z tych elementów jest duży, wówczas możemy już mówić o dużym systemie. Zazwyczaj, choć nie jest to warunek konieczny, w dużych systemach występuje również pewien poziom komplikacji logiki systemu, czyli tzw. **logika biznesowa**. Niektórzy postrzegają wielkość systemu przez pryzmat funkcji, jakie dostarcza on użytkownikowi końcowemu. Nie sądzę jednak, aby to podejście było słuszne. Świadczy o tym chociażby przykład wyszukiwarki Google, której interfejs oferuje użytkownikom niewiele funkcjonalności, natomiast z całą pewnością nie można o niej powiedzieć, że jest małym systemem. I wreszcie: duży system to taki, który jest krytyczny biznesowo dla klienta, czyli jego awarie i/lub błędne działanie powodują wymierne straty finansowe.

Zastanówmy się teraz, **w jakich obszarach mogą wystąpić problemy w dużych systemach.**

Po pierwsze, mogą mieć one źródło w kodzie **aplikacji**, który napisaliśmy. Drugi typ problemów dotyczy **oprogramowania aplikacyjnego**, z którego nasza aplikacja musi korzystać. Mowa tu o problemach występujących w serwerach http, serwerach aplikacyjnych, bazie danych, systemach kolejkowania czy systemach zewnętrznych, z którymi komunikuje się nasz system. Kolejny obszary, na których mogą wystąpić problemy, to **system operacyjny i protokoły sieciowe**, gdzie również czai się wiele

niespodzianek. I oczywiście jest jeszcze **sprzęt**, na którym to wszystko działa, a który również może być źródłem problemów.

Liczba błędów spada zgodnie z przedstawianą hierarchią elementów systemu – w aplikacji jest ich najwięcej, w sprzęcie najmniej. Natomiast im bliżej sprzętu znajduje się problem, tym jest on bardziej poważny i trudniejszy do rozwiązania.

Skoro już wiemy, gdzie mogą wystąpić problemy, zobaczmy, **co może nas zaboleć**.

Pierwsza rzecz to **stabilność systemu**. Użytkownicy oczekują, że w długim okresie czasu system będzie realizował pewne funkcje biznesowe, przynosząc im określone korzyści. System, który pracuje niestabilnie, automatycznie powoduje nieufność użytkowników końcowych, a tym samym jest powodem utraty wiarygodności przez właściciela systemu.

Kolejna sprawa to **wydajność**. Jest ona inaczej postrzegana od strony użytkownika końcowego niż od strony zamawiającego system. Użytkownik końcowy w większym stopniu oczekuje szybkiego czasu odpowiedzi, czyli czegoś, co określa się mianem *performance*. Natomiast klient zamawiający system będzie oczekiwać głównie przepustowości (*throughput*), czyli jak największej liczby transakcji biznesowych zrealizowanych w jednostce czasu. Proponuję zatem przyjąć, że wydajność jest to pewna przepustowość systemu przy akceptowalnym czasie odpowiedzi.

Jest również coś takiego jak **dostępność systemu**, określająca przez jaki czas system jest dostępny dla użytkowników. Jest ona związana w dużej mierze ze stabilnością i wydajnością.

I jeszcze dwa dodatkowe aspekty: **bezpieczeństwo**, którego znaczenie jest oczywiste, oraz **administrowanie systemem**. Zdarzają się systemy zbudowane w sposób, który znacząco utrudnia administrowanie. Przy pewnej skali systemu i obciążenia, czy też dużej liczbie serwerów, klastrów itd. zarządzanie takim systemem może być bardzo uciążliwe, a koszty jego utrzymania będą ekstremalnie wysokie.

Jeśli zestawimy te dwa aspekty, tzn. miejsce wystąpienia problemu oraz jego charakter, to otrzymamy całkiem sporą przestrzeń problemów (rysunek 1).

	Aplikacja	Oprogramowanie aplikacyjne	System operacyjny / protokoły sieciowe	Sprzęt
Stabilność	BUM!		BUM!	
Wydajność		BUM!		
Dostępność			BUM!	
Bezpieczeństwo	BUM!			
Administracja				BUM!

Rysunek 1. Przestrzeń problemów

Nie sposób jest w tym stosunkowo krótkim artykule omówić całą pokazaną powyżej przestrzeń problemów. Dlatego przedstawię tu trzy wybrane przykłady, z jakimi zetknęliśmy się w e-point SA.

Zacznę od rzeczy dosyć łagodnych, czyli **problemów ze współbieżnością aplikacji**.

Załóżmy, że mamy system, który po wdrożeniu pracuje stabilnie przez kilka miesięcy. Stale przybywa użytkowników, rośnie też obciążenie poszczególnych elementów systemu, ale pomimo tego wszystko działa raczej bezproblemowo. Do czasu. Dalszy wzrost liczby użytkowników powoduje pojawienie się niepokojących objawów – system zaczyna zamierać na kilkadziesiąt sekund, czasami na kilka minut. Część z tych przypadków kończy się całkowitą „śmiercią" systemu i jesteśmy zmuszeni go restartować. Przy czym, co jest istotne, w trakcie takiego „przymierania" obserwujemy dosyć mocny spadek obciążenia na procesorach, nie widzimy też jakichś intensywnych operacji I/O, czy to na bazie, czy na serwerze aplikacyjnym. I zaczynamy obserwować w serwerze aplikacyjnym – a konkretnie w logach aplikacji – wyjątki z zakleszczeniami (deadlock), czyli informację, którą zwraca nam pośrednio driver JDBC, że baza danych wykryła zakleszczenie transakcji, którą właśnie wykonywaliśmy, po czym ta transakcja została wycofana. Pojawiają się również wyjątki pokazujące, że transakcje JTA w serwerze aplikacyjnym timeout'ują. Analiza

sytuacji nie była prosta, ale w końcu odkryliśmy, jakie były przyczyny tego stanu rzeczy.

Po pierwsze, okazało się, że w dwóch konkretnych transakcjach biznesowych mieliśmy niekorzystny przeplot odczytów i zapisów. Chodziło o operacje na bazie danych. Już samo to mogło prowadzić do deadlocków, ale akurat w naszym przypadku musiało być coś jeszcze. I tym czymś było pełne skanowanie (full scan) jednej z tabel w bazie danych. Była to tabela, która występowała w obu transakcjach. Kiedy my żądaliśmy odczytu jednego rekordu, optymalizator uznawał, że szybciej będzie przeskanować całą tabelę. Z logicznego punktu widzenia działanie optymalizatora było poprawne, natomiast z punktu widzenia wydajności powodowało drastyczne obniżenie współbieżności, przez co kluczowe transakcję zaczynały się ze sobą blokować.

Kiedy już doszliśmy do tego, co powodowało problem, wówczas jego rozwiązanie nie było zbyt pracochłonne. Włącznie ze zmianą aplikacji i wgraniem na produkcję zajęło nam kilka godzin. Natomiast sama analiza i dojście do przyczyn problemu trwało kilka dni.

Rozwiązanie było proste. Operacje modyfikujące przenieśliśmy na koniec tych transakcji, co akurat było możliwe z punktu widzenia wymagań biznesowych. Następnie przygotowaliśmy specjalną podpowiedź dla optymalizatora bazy danych, która spowodowała, że preferowanym sposobem dostępu do tabeli było użycie indeksu, a nie pełne przeglądanie jej zawartości.

Podsumujmy teraz problemy ze współbieżnością.

Na uczelni mówiono nam, że jeśli mamy problem z deadlockami, to wystarczy w ustalonej kolejności blokować zasoby i wszystko wróci do normy, w szczególności nie będzie deadlocków. Niestety to rozwiązanie akademickie działa tylko teoretycznie, natomiast jest trudno realizowalne w praktyce, nawet w małych systemach. Dlaczego? Przede wszystkim, w realnym systemie biznesowym mamy do czynienia z tysiącami transakcji, które nawzajem się przeplatają i każda z tych transakcji zajmuje pewne zasoby – zazwyczaj kilka, kilkanaście, a czasami nawet kilkadziesiąt. I, co gorsza, nie mamy praktycznie żadnej kontroli nad blokowaniem tych zasobów, ponieważ zazwyczaj operujemy na bazie danych. To baza danych decyduje w dużej mierze o tym, które zasoby zablokować. Jeżeli chcemy pobrać

jeden wiersz z określonej tabeli, to zazwyczaj oczekujemy, że tylko on zostanie zajęty. Ale niekoniecznie musi to być prawda. Równie dobrze może to być strona tabeli (wiersze w bazie danych grupowane są zazwyczaj w większe jednostki danych) lub nawet cała tabela. Jeżeli nie mamy tej kontroli, to absolutnie nie ma mowy o ustalaniu jakiejś kolejności, bo i tak to nic nie da. Czasami też z wymagań biznesowych dla poszczególnych transakcji wynika, że nie można odwrócić pewnych działań w transakcji, co oczywiście już w pierwszym podejściu rozkłada akademickie podejście na łopatki.

Jeśli jest tak źle, to co możemy zrobić w praktyce? Przede wszystkim możemy testować system za pomocą realnych scenariuszy biznesowych. Chodzi o to, aby po wdrożeniu systemu przyglądać się aktywności użytkowników i na podstawie uzyskanych w ten sposób danych budować scenariusze biznesowe. W ten sposób, przy wprowadzaniu zmian w aplikacji lub rozbudowie systemu, możemy w warunkach laboratoryjnych realizować określone scenariusze, mocno obciążając system i obserwując wszystkie parametry związane ze współbieżnością – czyli to, co dzieje się na bazie danych oraz w serwerze aplikacyjnym. Niezwykle istotne jest również monitorowanie parametrów związanych ze współbieżnością na bieżąco w trakcie działania systemu produkcyjnego.

Kolejny temat to **styk z systemami zewnętrznymi**. Przyjmijmy, że mamy dojrzały system pracujący stabilnie od dwóch lat. Wszystko działa poprawnie, nie ma żadnych problemów wydajnościowych, użytkownicy są zadowoleni. Na życzenie klienta wdrażamy funkcję potwierdzania transakcji SMS-em. Po tym wdrożeniu system pracuje stabilnie przez kilka tygodni i nagle staje się niedostępny dla użytkowników. Po naszej stronie mamy całkowity spadek obciążenia na wszystkich elementach systemu. W systemie nie dzieje się nic, jeśli nie liczyć brzegowego serwera http przyjmującego liczne nieudane żądania. Tym, co obserwujemy, jest wysycenie puli wątków w serwerach aplikacyjnych. Wszystkie wątki w serwerze, które mogły przetwarzać żądania, są zajęte. W logach cicho. Co próbujemy zrobić?

Ponieważ na pierwszy rzut oka nic nie można z tego wywnioskować, więc decydujemy się na restart systemu. System wstaje, ale po 2 minutach ponownie pada. Więc restartujemy jeszcze raz i znowu to samo. Pomaga dopiero wyłączenie modułu do wysyłania SMS-ów. System wstaje i działa poprawnie. Wiemy już, gdzie leży problem. Okazało się, że nowa funkcjonalność zepsuła coś w sposób dosyć

drastyczny. Po przeanalizowaniu sytuacji odkryliśmy, że bezpośrednią przyczyną problemu była niedostępność bramki SMS-owej. Przy czym bramka nie odrzucała połączenia, tylko po prostu nic nie działo się z wysłanymi przez nas żądaniami, które tam sobie najzwyczajniej w świecie „wisiały”. Timeouty TCP na poziomie systemu operacyjnego są bardzo długie, więc w tym czasie nic nie zdążyło się zerwać i nie mieliśmy żadnych wyjątków. Prawdziwa przyczyna problemu leżała gdzie indziej. Wynikała ona z niedostatecznej separacji naszego systemu i bramki SMS. Pewna separacja została przez nas przewidziana, ale okazała się niewystarczająca. W naszym rozwiązaniu, gdy użytkownik wykonywał transakcję, to jej dane trafiały do odpowiedniej struktury w bazie danych, a dodatkowo do oddzielnej tabeli trafiał komunikat SMS, który był następnie pobierany i wysyłany przez całkowicie asynchroniczny proces. Niestety proces ten powodował blokowanie do zapisu tabeli z komunikatami, co w przypadku problemów z bramką SMS skutkowało zawieszeniem pracy całego systemu.

Jakie mamy tutaj rozwiązania?

Tego typu problemy rozwiązuje się zazwyczaj przez wprowadzenie kolejkowania. Może to być np. JMS. Ponadto, obowiązkowe jest wprowadzenie timeoutów przy komunikacji z takimi systemami zewnętrznymi. I to timeoutów na różnych poziomach, nie tylko na poziomie logicznym, ale również na poziomie TCP. My akurat nie mogliśmy zastosować kolejkowania, więc musieliśmy użyć triku z bazą danych. Po prostu zreorganizowaliśmy dostęp do bazy danych w taki sposób, aby odczyt z tabeli komunikatów nie blokował nam żadnych zapisów, które mogą tam trafić.

Styk z systemami zewnętrznymi, z którymi się integrujemy lub współdziałamy, jest najczęstszą przyczyną problemów w dużych systemach. Zawsze na tych interakcjach, gdzie w grę wchodzi sieć, bardzo dużo może się wydarzyć. Kolejne przykłady takich punktów styku to:

- **Baza danych** – bardzo często uważamy, że baza danych jest integralną częścią systemu, ale tak naprawdę – jeśli patrzymy z punktu widzenia aplikacji – jest to taki sam punkt styku, jak w przypadku systemu backend’owego czy bramki SMS. Dlatego też musimy dobrze skonfigurować odpowiednie timeout’y i parametry pól połączeń, aby „nie iskrzyło” na styku między serwerami aplikacyjnymi a bazą danych.

- **Serwery poczty** – które też czasami niedomagają w specyficzny dla siebie sposób.
- **Systemy zewnętrzne** – współcześnie budowane systemy praktycznie nigdy nie istnieją niezależnie od innych systemów zewnętrznych. Zawsze jest z boku jakiś inny – mniejszy lub większy – system, który uczestniczy w pracy naszego systemu.
- **Podsystem logowania zdarzeń** – ciekawostką jest, że on też może dać nam w kość. Znam przykład serwera aplikacyjnego, który logował zdarzenia, a gdy log urósł do rozmiarów 2 GB serwer nagle zniknął, tzn. zniknął JVM i nie dał się później uruchomić. Rozwiązaniem było wyczyszczenie loga, ale żeby dojść do tego rozwiązania, jakiś czas należało mocno pogłówkować.

Wydawałoby się, że **pule wątków** nie wysycają się tak szybko, zwłaszcza jeśli mamy sporej wielkości rozwiązanie typu 10 maszyn wirtualnych po 50 wątków w każdej maszynie. To daje nam 500 wątków, które tylko czekają, żeby przyjąć ruch z Internetu i obsłużyć użytkowników. Gdy jednak w systemie obsługiwanych jest jednocześnie 100 requestów na sekundę i poszczególne wątki zaczną się nawzajem blokować, to kwestią tylko kilku lub kilkudziesięciu sekund, czasami 2 minut, jest zablokowanie wszystkich wątków, czego skutkiem jest totalna zapaść systemu.

Na koniec rozważań o problemach z systemami zewnętrznymi chciałbym zwrócić uwagę na **warstwę sieci**. Tu niestety bardzo dużo może się wydarzyć. Stosy TCP/IP w systemach operacyjnych, firewall'e, rutery, switch'e, kanały VPN do systemów, z którymi w jakiś sposób się integrujemy... Wszystko to może przestać działać i to, jak zwykle, w najmniej oczekiwanym momencie. Gorsza sprawa, że może tylko udawać, że działa, co w przypadku sieci jest częstym zjawiskiem. Do tego dochodzą awarie typu uszkodzony kabelek, który trochę przepuszcza, ale nie do końca.

Kolejny temat to **pamięć w serwerach aplikacyjnych jako ograniczenie wydajnościowe**. I od razu „mocny" przykład. Jak zwykle zacznę od opisu sielanki: mamy system stabilnie pracujący od kilkunastu miesięcy, liczba użytkowników systematycznie rośnie, system jest trochę bardziej obciążony, ale to niespecjalnie daje się we znaki. Na serwerze aplikacyjnym 70% zużycia procesora. Zamawiający jest zadowolony. Jednak w momentach zwiększonego ruchu zaczynamy obserwować niepokojące objawy. Na początku coraz dłuższy czas odpowiedzi

systemu i bardzo często maksymalne obciążenie procesora na serwerach aplikacyjnych, które czasami doprowadzają do załamania systemu. W logach nie obserwujemy niczego wyjątkowego, więc wydaje się, że są to klasyczne objawy zwykłego przeciążenia systemu. Po prostu Java wypala procesor i wypadałoby dołożyć więcej mocy.

Mamy tutaj dwie metody rozwiązania tego klasycznego problemu z wydajnością Javy. Możemy jeszcze bardziej zoptymalizować już zoptymalizowaną aplikację, co wymaga zmian w kodzie i późniejszych testów. Stanowi więc pewne ryzyko. Ale przede wszystkim podnosi późniejsze koszty eksploatacji systemu, o czym nie wszyscy pamiętają. Drugie rozwiązanie to dołożenie sprzętu, co wydaje się lekkie, łatwe i przyjemne. Można to zrobić szybko – postawić kolejną maszynę, zainstalować odpowiednie oprogramowanie, dołączyć maszynę do klastra i oczekiwać pozytywnych efektów.

My wybraliśmy tę drugą drogę. Podłączyliśmy kolejny sprzęt i faktycznie odnotowaliśmy poprawę, ale nie taką, jakiej się spodziewaliśmy, czyli daleką od liniowej. Na nowo przeanalizowaliśmy problem. Skoro Java tak „paliła", to zaczęliśmy bardziej wnikliwie przyglądać się pracy maszyny wirtualnej. I co się okazało? Przyczyną problemu nie była niewystarczająca moc procesora, tylko niewystarczająca ilość pamięci dla aplikacji i w związku z tym zbyt częste uruchamianie procesu odśmiecania pamięci.

Tak się składa, że współczesne aplikacje mają tendencję do alokowania dużej liczby nowych obiektów, a następnie zostawiana ich „na pożarcie" maszynie wirtualnej. Na każdym żądaniu tworzymy bardzo dużo obiektów i potem liczymy na to, że zostaną w elegancki sposób posprzątane. W przypadku maszyny wirtualnej, z którą mieliśmy do czynienia, tak się nie działo i przysłowiowy „garbaty" bardzo często musiał pracować. Tak naprawdę procesor głównie obsługiwał proces odśmiecania, zamiast wykonywać kod aplikacji.

Jakie rozwiązanie zastosowaliśmy? Przede wszystkim znacząco rozbudowaliśmy pamięć fizyczną w serwerach i uruchomiliśmy na każdym z serwerów fizycznych kilka maszyn wirtualnych. Dlaczego tak, a nie inaczej? Dlaczego np. nie zwiększyliśmy pamięci na maszynie wirtualnej? Ponieważ była to maszyna 32-bitowa i po prostu nie dało się tego zrobić. Na tym systemie operacyjnym, z którym mieliśmy do czynienia,

mogliśmy pamięć sterty powiększyć tylko do tysiąca sześciuset megabajtów, czyli niecałych 2 GB. Więcej się nie dało, zatem trzeba było to rozwiązać inaczej.

Ponieważ przypadek jest dosyć ciekawy, poddam go szczegółowej analizie.

Przede wszystkim: co znajduje się na stercie pamięci w serwerze aplikacyjnym, gdzie nasza aplikacja jest zainstalowana? W zasadzie są tam dwa typy obiektów. Obiekty długo żyjące, związane z serwerem aplikacyjnym, z sesją użytkowników, pamięcią podręczną, które podczas procesu odśmiecania pamięci nie podlegają usunięciu z pamięci, albo też dzieje się to bardzo rzadko, np. dla sesji użytkownika. Z drugiej strony, mamy obiekty krótko żyjące, czyli w zasadzie wszystkie te, które są związane z obsługą każdego żądania przesyłanego do serwera. Te obiekty, praktycznie przy każdym uruchomieniu garbage collectora, są usuwane z pamięci systemu.

Dla wydajności całego systemu istotny jest czas pracy poświęcony na proces odśmiecania (zmienna `Garbage Collector time`, GC_{time}) oraz czas, jaki występuje pomiędzy poszczególnymi uruchomieniami garbage collector'a (zmienna `Allocation Failure Distance`, $AF_{distance}$, określająca jak często on się uruchamia).

Jak teraz określić zużycie procesora dla aplikacji i dla garbage collectora? W pierwszym przypadku bierzemy po prostu czas odśmiecania (GC_{time}) i dzielimy go przez sumę czasu odśmiecania i odstępów pomiędzy uruchomieniami garbage collectora ($AF_{distance}$), czyli przez całkowity czas, który jest potrzebny na odśmiecanie pamięci i normalną pracę (patrz: rys. 2).

$$CPU_{GC}\,[\%] = \frac{GC_{time}}{GC_{time} + AF_{distance}}$$

Rysunek 2.

W przypadku aplikacji mamy zależność odwrotną, czyli czas procesora poświęcony na działanie aplikacji ($AF_{distance}$) dzielimy przez czas, w którym nie działa garbage collector (patrz: rys. 3).

$$CPU_{APP}\,[\%] = \frac{AF_{distance}}{GC_{time} + AF_{distance}}$$

Rysunek 3.

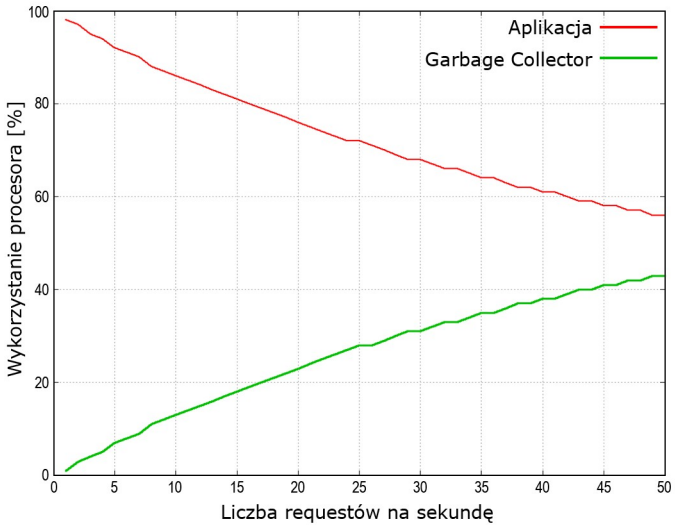
Jeśli chodzi o czas odśmiecania, to z dużym przybliżeniem można powiedzieć, że jest on wprost proporcjonalny do wielkości pamięci zajętej przez obiekty długo żyjące. Przynajmniej taką właśnie zależność obserwujemy dla algorytmu *mark and sweep*.

Natomiast odległość pomiędzy poszczególnymi uruchomieniami procesu odśmiecania ($AF_{distance}$) możemy określić odejmując wielkość obiektów długo żyjących (`LongLivedObjectsSize`) od wielkości sterty ustalonej na serwerze (`HeapSize`) i dzieląc wszystko przez iloczyn liczby requestów na sekundę (`RPS`) oraz wielkości obiektów krótko żyjących generowanych przez każdy request (`ShortLivedObjectsSizePerRequest`). Zależność tę ilustruje rys. 4.

$$AF_{distance} = \frac{HeapSize - LongLivedObjectsSize}{RPS * ShortLivedObjectsSizePerRequest}$$

Rysunek 4.

Cofnijmy się teraz do przykładu naszej aplikacji, która ma do dyspozycji stertę wielkości około 2GB. Zakładamy, że idą kolejne requesty. Okazuje się, że w naszym przypadku pamięć na obiekty długo żyjące wynosi ok. 700 MB, co stanowi mniej więcej 1/3 sterty. Na każdy request przypada nieco ponad 1 MB. Doświadczalnie wyznaczamy, że czas odśmiecania tego 1 MB sterty zajmuje jakieś 2-3 milisekundy. Korzystając z wcześniej przedstawionej zależności możemy wyznaczyć czas wykorzystania procesora na pracę aplikacji i na proces odśmiecania pamięci, co ilustruje rysunek 5.



Rysunek 5. Czas procesora poświęcony na pracę aplikacji i proces odśmiecania pamięci

Przy 50 requestach proces odśmiecania zajmuje 40% czasu procesora, czyli – delikatnie mówiąc – niewiele nam pozostaje na pracę aplikacji. I teraz cały trik polega na tym, aby maksymalnie spłaszczyć tę krzywą. Jeżeli nie mamy możliwości dokonania żadnego manewru w maszynie wirtualnej (np. zmiany algorytmu odśmiecania lub „pokręcenia" innymi parametrami), to najprostszym sposobem jest dołożenie pamięci i spowodowanie, żeby przestrzeń na obiekty krótko żyjące była dużo większa. W naszym przypadku dokładnie tak zrobiliśmy.

Na koniec artykułu proponuję przyjrzeć się z bliska tematowi **dostępności** czyli mitycznym dziewiątkom…

W niektórych dokumentacjach do serwerów aplikacyjnych opisane są scenariusze, jak osiągać kolejne dziewiątki – pierwszą, drugą, trzecią… piątą… a nawet ósmą! Zobaczmy, czy w praktyce da się zapewnić klientowi te kolejne dziewiątki, a jeśli tak, to jak to zrobić.

Pierwsza dziewiątka – dostępność systemu na poziomie **90%**. Oznacza to 73 godziny niedostępności w miesiącu. Mamy więc dużo czasu, aby coś naprawić, czy wręcz kupić nowy sprzęt w supermarkecie i zainstalować wszystko od nowa lub

skorzystać z dowolnego centrum hostingowego. Aby wytworzyć taki system, zespół nie musi posiadać zaawansowanej wiedzy, a z jego obsługą poradzi sobie nawet dochodzący administrator. Bułka z masłem :-)

Zobaczmy, co się stanie, gdy dołożymy drugą dziewiątkę. Otóż **99%** dostępności to nieco ponad 7 godzin w zapasie. Zaczynają się schody. Po pierwsze, sprzęt z supermarketu już nie wystarczy. Niezbędne minimum, jakie musimy zagwarantować, to serwer zapasowy, przygotowany tak, aby w każdej chwili zastąpić ten, który potencjalnie może się zepsuć na produkcji. Nie mamy bowiem czasu, aby wszystko instalować od zera. Kolejna sprawa: należy zadbać o porządne wykonanie pewnych elementów systemu, np. przemyśleć aspekty współbieżności, interakcje z innymi systemami – tak, aby wszystko składało się w jedną całość. I wreszcie: taki poziom dostępności wymaga nadzoru w systemie 24/7. Czy zatem jeden administrator poradzi sobie z tym zadaniem? Powiedzmy, że tak – że są tacy, którzy w pojedynkę daliby radę, ale z pewnością nie obejdą się bez mechanizmów automatycznego monitorowania systemu, które na tym poziomie są już niezbędne. Nadal jednak, mimo pewnych trudności, osiągnięcie dwóch dziewiątek nie jest szczególnie trudne.

Pójdźmy dalej i skomplikujmy sytuację dokładając kolejną dziewiątkę. **99,9%** dostępności daje nam 43 minuty w miesiącu na awarie. I to jest już, moim zdaniem, wyzwanie wyłącznie dla profesjonalistów. Przede wszystkim niezbędna jest pełna redundancja sprzętu, oprogramowania systemowego i aplikacyjnego. To znaczy musimy zapewnić sobie klastry wydajnościowo niezawodne – zarówno w serwerach http, serwerach aplikacyjnych, jak i w serwerach bazy danych. Oczywiście to samo dotyczy sprzętu – musimy mieć zapewnione klastrowane switch'e, firewall'e, macierze dyskowe itd. Potrzebny jest też bardzo staranny projekt i precyzyjna implementacja takiego systemu, ponieważ nie ma tu już miejsca na większą awarię. Kolejna sprawa to analiza stanów przedawaryjnych – system należy przez cały czas monitorować i szybko reagować na potencjalne problemy. Ważna jest bowiem stała prewencja i zapobieganie, a nie tylko gaszenie pożaru już po wystąpieniu awarii. Do tego wszystkiego musimy zapewnić bardzo doświadczone zespoły – deweloperski i hostingowy – które uczestniczyły już w podobnych zadaniach. Muszą one pracować w trybie ciągłym, na bieżąco monitorować system i szybko reagować nawet na najmniejsze symptomy zbliżającej się awarii. No

i musimy wyposażyć administratorów w zautomatyzowane procedury naprawcze. 43 minuty to naprawdę niewiele i jeśli weźmiemy pod uwagę stres, który dochodzi w przypadku awarii, to okazuje się, że w tak krótkim czasie człowiek nie jest w stanie zrobić wiele więcej niż nacisnąć guzik stop/start. Przy czym implementacja takiego guzika w dużych systemach jest mocno nietrywialna. Wyłączenie i postawienie od zera systemu, który pracuje na kilkudziesięciu maszynach i składa się ze 100 czy 200 komponentów softwarowych, zajmuje zazwyczaj parę minut, nawet jeśli jest on dobrze wykonany. Wymagane jest również przygotowanie systemu do rekonfiguracji w locie. Dotyczy to zarówno samej aplikacji, jak i komponentów, na których ta aplikacja pracuje. Co jeszcze? Przydałoby się również podwójne monitorowanie, a najlepiej monitorowanie monitoringu. Chociaż, być może, jest to warunek bardziej adekwatny dla kolejnej dziewiątki. A podsumowując temat trzech dziewiątek: z mojego doświadczenia wynika, że to jeszcze da się osiągnąć.

Zatem doszliśmy do czwartej dziewiątki i tu… niespodzianka – nie opiszę, jak ją osiągnąć, ponieważ sam nie potrafię tego zrobić. **99,99%** oznacza mniej niż 5 minut niedostępności w miesiącu, a więc takie trochę *mission impossible*. Niestety klienci czasami tego właśnie oczekują, a najbardziej chcieliby mieć 100% dostępności i to na dodatek za darmo. Takie podejście klientów można jeszcze zrozumieć. Natomiast zaskakujące jest, że dostawcy oprogramowania J2EE obiecują nam nie tylko cztery, ale nawet pięć czy więcej dziewiątek! Przedstawiają w dokumentacji wspaniałe scenariusze, wielkie wykresy, rozbudowane diagramy różnych sprzętów, klastrów i, Bóg wie, czego jeszcze. I obiecują właśnie te osławione pięć dziewiątek czy też więcej – czasami osiem, co dla mnie jest już czystą abstrakcją.

Dlaczego uważam, że w praktyce jest to niewykonalne? Ponieważ nawet najlepsze centra hostingowe dostarczają obecnie usługi z dostępnością 99,95%, czyli zostawiają sobie w zapasie 21 minut na własne awarie. A przecież trzeba jeszcze doliczyć czas na ewentualne awarie po naszej stronie. Widać więc, że tego za bardzo nie da się zrobić, nawet gdyby poszczególne komponenty działały na poziomie dostępności 99,99% czy nieco więcej.

Zastanówmy się teraz, ile może kosztować każda kolejna dziewiątka? Według obliczeń dokonanych za oceanem, każda następna dziewiątka zwiększa koszt wytworzenia systemu o rząd wielkości i podwaja roczny koszt jego utrzymania. Według moich szacunkowych kalkulacji, ta prawidłowość działa co najmniej

w przypadku trzech pierwszych dziewiątek. Zatem decyzja o planowaniu dostępności systemu musi być decyzją biznesową, poprzedzoną szczegółowymi wyliczeniami. Każda firma powinna sobie indywidualnie obliczyć, ile może ponieść straty, a ile zaoszczędzić na tym, że system będzie bardziej lub mniej dostępny.

Na sam koniec małe podsumowanie.

Bardzo często spotykamy się z takim twierdzeniem, że TO nie ma prawa się zdarzyć w przypadku naszego systemu, a prawdopodobieństwo, że TO nas spotka, można mierzyć w skali niemalże kosmicznej. Nic bardziej błędnego. Drobne obliczenie. W systemie internetowym średniej skali mamy około 500 odsłon na sekundę, każdy użytkownik „ciągnie" 10 elementów na stronie – powiedzmy flash'e i inne ozdobniki. Mamy 3600 sekund w godzinie, 24 godziny na dobę, 365 dni w roku, a system utrzymujemy przez 5 lat – bo taki jest średni czas życia systemu internetowego. I dochodzimy do wniosku, że mamy prawie 800 miliardów szans na to, aby coś poszło źle. W Drodze Mlecznej, według ostatnich obliczeń, jest 400 miliardów gwiazd, więc skala kosmiczna jest tu jak najbardziej adekwatna. To oznacza, że problemy (takie odpowiedniki wybuchu supernowej) mogą nam się przytrafiać codziennie.

Zasadnicze pytanie nie jest więc takie, czy coś się może wydarzyć, ale kiedy to coś się wydarzy. I trzeba być na to przygotowanym. W dużych systemach, mam nadzieję, że udało mi się to pokazać, mamy do czynienia z naprawdę dużymi problemami, ale jest też ogromna satysfakcja, jeśli te problemy udaje nam się pokonać. Sądzę, że jest sporo osób, szczególnie z dłuższym stażem deweloperskim, które znajdują prawdziwą satysfakcję z rozwiązywania problemów właśnie w takich systemach. Jestem jedną z nich.

Masz pytanie do autora? Napisz:
jaroslaw.blad@e-point.pl