STUDIA INFORMATICA 2015
Volume 36 Number 1 (119)

Łukasz WARCHAŁ
Asseco Poland S.A.
Dariusz R. AUGUSTYN, Łukasz WYCIŚLIK
Politechnika Śląska, Instytut Informatyki

KONCEPCJA ARCHITEKTURY ROZPROSZONEGO SYSTEMU KLASY HIS DLA SIECI JEDNOSTEK OCHRONY ZDROWIA[1]

Streszczenie. Procesy konsolidacyjne przebiegające w służbie zdrowia oraz nieustająca presja na ograniczanie kosztów (w tym kosztów IT) wymuszają na dostawcach oprogramowania dla szpitali stosowanie nowoczesnych rozwiązań, szczególnie w obszarze architektury systemu. Artykuł opisuje koncepcję architektury systemu klasy HIS opartą na mikrousługach, która zapewnia stabilność, wydajność i łatwość skalowania, co jest kluczowym zagadnieniem w przypadku wdrożenia systemu w sieci jednostek ochrony zdrowia.

Słowa kluczowe: architektura systemów, mikrousługi, systemy medyczne, skalowalność

CONCEPTION OF ARCHITECTURE FOR DISTRIBUTED HIS SYSTEM FOR FEDERATED HOSPITALS

Summary. Consolidation processes ongoing in polish healthcare sector combined with continuous pressure for cost reduction (including IT costs) forces software vendors to use modern technologies, especially in software architecture area. This article describes concept of architecture for HIS system based on micro-services architecture, which ensures stability, performance and scalability, which is crucial in case of deployment the solution in federated hospitals.

Keywords: system architecture, micro-services, HIS systems, scalability

[1] This work was supported by NCBiR of Poland (No INNOTECH-K3/IN3/46/229379/NCBR/14).

1. Wstęp

W obecnej rzeczywistości trudno wyobrazić sobie działanie szpitala bez sprawnego systemu informatycznego. Zarówno wielkie kliniki, jak i mniejsze szpitale powiatowe muszą zapewnić swoim lekarzom i pracownikom administracyjnym dostęp do komputerów i innych urządzeń zaopatrzonych w specjalistyczne oprogramowanie, aby mogli oni wykonywać swoje codzienne obowiązki na właściwym poziomie. Rozwój technologii i powszechny dostęp do sieci Internet sprawiają, że również pacjenci zaczynają wchodzić w interakcję ze szpitalem nie tylko w świecie rzeczywistym, ale także w wirtualnym. Stają się oni de facto kolejną grupą użytkowników systemu informatycznego placówki ochrony zdrowia. Rośnie więc liczba osób korzystających z różnego rodzaju aplikacji i systemów, co w oczywisty sposób zwiększa koszty działalności szpitala. Stąd nieustająca presja na obniżanie wydatków związanych z utrzymaniem szeroko rozumianego IT i próby przenoszenia tego fragmentu działalności szpitala do podmiotów zewnętrznych (ang. outsourcing), które oferują rozwiązania, z których może korzystać kilka jednostek [8].

W tych realiach dostawcy oprogramowania klasy HIS (ang. *Hospital Information System*) stoją przed nie lada wyzwaniem. Muszą bowiem wytworzyć produkt, który zaspokoi bardzo różne potrzeby i oczekiwania swoich odbiorców, które dodatkowo często się zmieniają, np. wskutek zmian w prawie. Oferowany produkt musi być dopasowany do infrastruktury małego szpitala, ale musi się go także dać zainstalować w centrum przetwarzania, które obsługuje sieć dużych szpitali. Bardzo ważną kwestią jest wysoka wydajność oraz odporność na awarie. Przerwy w działaniu powinny być jak najkrótsze, aby nie wstrzymywać pracy szpitala. Całe rozwiązanie musi zapewniać najwyższy poziom bezpieczeństwa, ze względu na charakter przetwarzanych informacji (wrażliwe dane medyczne). W większości szpitali oprogramowanie klasy HIS nie jest jedynym działającym w nim systemem informatycznym i musi integrować się z innymi aplikacjami, aby zapewnić pełne wsparcie realizowanych procesów biznesowych.

Architektura trójwarstwowa czy ta typu klient - serwer to obecnie najczęściej używane architektury systemów klasy HIS [17]. Cechuje je zwarta i monolityczną budowa [7], przez co elastyczne skalowanie czy wdrożenie w centrum przetwarzania danych (chmura obliczeniowa) jest problematyczne.

Artykuł przedstawia propozycję nowoczesnej architektury rozproszonego systemu klasy HIS, która w możliwie największym stopniu wspomaga procesy biznesowe realizowane przez sieć jednostek ochrony zdrowia (szpitale), w których system może być wdrożony. Praca zawiera wiele zaleceń, dotyczących wytwarzania poszczególnych składników systemu (komponentów), opisuje mechanizmy komunikacji między nimi oraz możliwe warianty

wdrożenia całego oprogramowania. W artykule opisano także technologie i narzędzia, dzięki którym możliwa jest implementacja zaproponowanego rozwiązania.

2. Koncepcja architektury rozproszonego systemu klasy HIS

2.1. Podstawowe założenia

Architektura systemów informatycznych definiowana jest jako zbiór reguł oraz podjętych decyzji, wpływających na kształt oprogramowania, na sposób komunikacji między składnikami aplikacji, na to jak wymienia ona dane ze światem zewnętrznym itp. [2, 3]. Kolejne podrozdziały opisują podstawowe założenia architektury rozproszonego systemu klasy HIS dla sieci szpitali.

2.1.1. Wielowarstwowość

Koncepcja warstw jest obecna w inżynierii oprogramowania niemal od samego początku. Warstwy są wykorzystywane zarówno w modelowaniu protokółów sieciowych (model ISO/OSI), jak i złożonych systemów działających w przedsiębiorstwach (architektura wielowarstwowa).

System klasy HIS obsługujący sieć szpitali powinien składać się z trzech głównych warstw:

- warstwy klientów,
- warstwy przetwarzania,
- warstwy przechowywania.

Warstwa klientów to głównie aplikacje GUI oraz zewnętrzne systemy oraz urządzenia. Komunikują się one z komponentami warstwy przetwarzania poprzez sieć, przy czym aplikacje posiadające graficzny interfejs użytkownika (obsługiwane przez lekarzy, pracowników administracyjnych i pacjentów) wykorzystują protokół HTTP/S, a systemy i urządzenia mogą także używać wprost TCP/IP. Warstwa przetwarzania zawiera algorytmy i reguły przetwarzające dane pozyskane od użytkowników lub z innych systemów. Przetworzone dane są zapisywane i odczytywane ze źródeł danych osadzonych w warstwie przechowywania.

Komunikacja pomiędzy komponentami z różnych warstw odbywa się przy użyciu jasno określonych interfejsów.

2.1.2. SOA

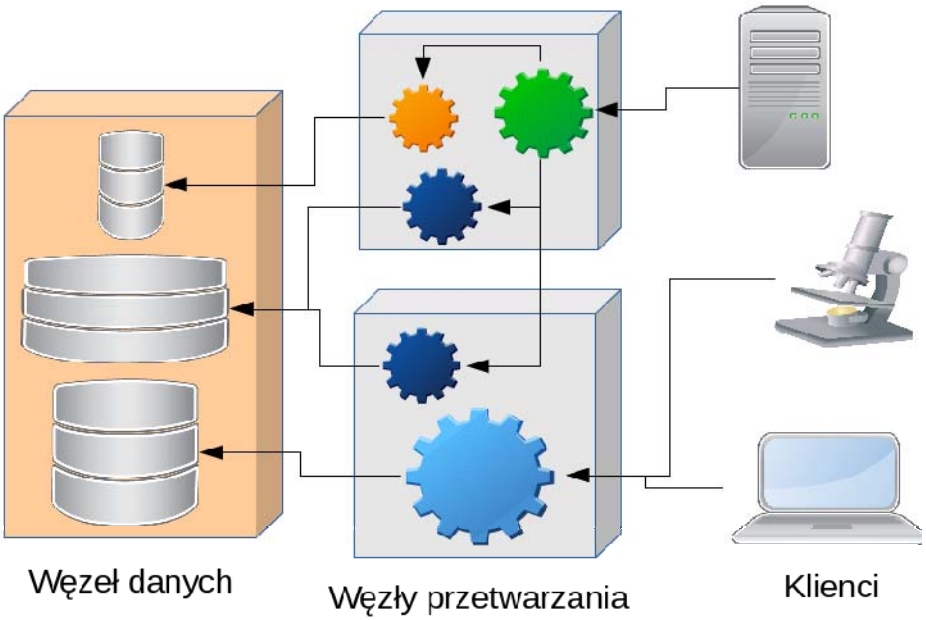
Architektura zorientowana na usługi (ang. Services Oriented Architecture) to jedna z najlepiej ugruntowanych koncepcji architektury systemów informatycznych ostatnich lat [4].

W jej rozumieniu usługa to autonomiczny, możliwy do użycia w różnych kontekstach (ang. *reusable*) komponent aplikacji. Na wysokim poziomie abstrakcji architektura warstwy przetwarzania systemu klasy HIS, obsługującego sieć szpitali, powinna być zgodna z paradygmatami SOA, tzn. jej komponenty powinny oferować bezstanowe usługi, możliwe do wykorzystania w różnych kontekstach, o jasno określonej odpowiedzialności. Każda usługa powinna realizować fragment lub całość procesu biznesowego realizowanego w jednostce ochrony zdrowia.

2.1.3. System rozproszony (mikrousługi)

Wdrożenie systemu klasy HIS w sieci jednostek ochrony zdrowia może przebiegać na kilka sposobów [8]. Dwa najbardziej charakterystyczne to: współdzielenie przez szpitale infrastruktury we wspólnym centrum przetwarzania danych, w którym instalowane są dedykowane dla każdej jednostki systemy HIS, które następnie integrują się ze sobą; instalacja jednego, zintegrowanego systemu dla wszystkich współpracujących jednostek. Z analizy ww. wariantów wdrożenia wynika, że w każdym przypadku system będzie musiał sprostać większemu obciążeniu. W związku z tym możliwość elastycznego skalowania staje się kluczowa. Spełnienie tego wymagania wymusza odpowiednią dekompozycję warstwy przetwarzania systemu HIS na wspomniane wcześniej usługi oraz rozproszenie ich pomiędzy dostępne w danej chwili węzły obliczeniowe. Ich liczba jest zależna od liczby użytkowników i przyjętego wariantu wdrożenia. W konsekwencji architektura systemu HIS dla sieci szpitali powinna zmienić się z monolitycznej w architekturę opartą na mikrousługach (ang. *microservices architecture*) [5, 6, 2]. Każda mikrousługa może być niezależnie uruchomiona na wybranych węzłach obliczeniowych. W celu zapewnienia maksymalnej skalowalności powinna posiadać własną składnicę danych. Wszelka komunikacja z innymi mikrousługami lub zewnętrznymi aplikacjami odbywa się poprzez sieć. Rysunek 4 przedstawia przykład takiej architektury.

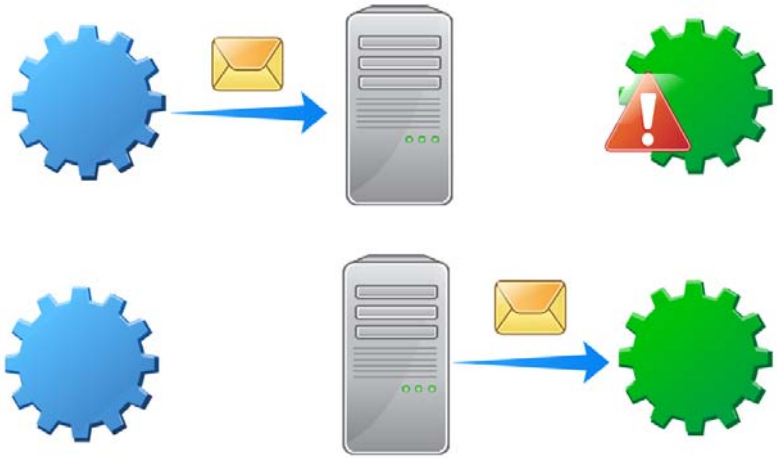
Zastosowanie architektury opartej na mikrousługach rozwiązuje także problem aktualizacji systemu – poszczególne jego fragmenty mogą być aktualizowane niezależnie, nie powodując zatrzymania całości. Możliwe jest także równoległe działanie kilku wersji danej usługi, co ma największe znaczenie w przypadku wdrożenia systemu w chmurze (centrum przetwarzania), gdy np. korzystające z niego szpitale mają wykupioną licencję na różne jego wersje.



Rys. 1. Schemat architektury opartej na mikrousługach
Fig. 1. Diagram of micro-services based architecture

2.1.4. Komunikacja asynchroniczna

W systemie o klasycznej, monolitycznej budowie poszczególne komponenty komunikują się poprzez proste wywołania metod (komunikacja wewnątrzprocesowa). Architektura oparta na mikrousługach wymaga większej uwagi w projektowaniu mechanizmów komunikacji, ponieważ wszelka wymiana danych odbywa się poprzez sieć (komunikacja międzyprocesowa) [9].



Rys. 2. Schemat komunikacji asynchronicznej z tzw. brokerem komunikatów
Fig. 2. Diagram of an asynchronous communication with a message broker

Niezależnie od wybranej architektury istnieją dwa podstawowe typy komunikacji: synchroniczna i asynchroniczna. Ta druga szczególnie dobrze nadaje się do dużych, rozproszonych systemów informatycznych, gdyż nie ogranicza skalowalności rozwiązania. Asynchroniczna wymiana danych pomiędzy dwoma składnikami systemu najczęściej realizowana jest

z wykorzystaniem dodatkowego komponentu pośredniczącego (tzw. brokera wiadomości), który jest odpowiedzialny za dostarczenie komunikatu do adresata. Jeżeli jest on chwilowo niedostępny (np. z powodu awarii czy aktualizacji), broker przechowuje wiadomość do czasu, aż adresat stanie się znów osiągalny. Schemat takiej komunikacji prezentuje rysunek 5.

Asynchroniczna wymiana komunikatów sprawia, że system jest bardziej reaktywny [11], poprzez większą odporność na awarie i zwiększoną wydajność. Zapewnia także odseparowanie poszczególnych jego elementów, co ułatwia ich autonomiczne testowanie w procesie produkcji.

System klasy HIS dla sieci szpitali powinien stosować asynchroniczną komunikację wszędzie, gdzie jest to możliwe. Jednak w wielu przypadkach zdarza się, że składniki systemu muszą wymienić dane w sposób synchroniczny, ponieważ wymaga tego specyfika procesu biznesowego. Wtedy komunikacja powinna odbywać się za pomocą usług sieciowych typu RESTful lub SOAP dostępnych poprzez protokół HTTP/S. Zalecany jest styl RESTful, gdyż jest on w pełni zgodny z mechanizmami sieci Internet [10], która jest podstawą działania systemu obsługującego wiele jednostek ochrony zdrowia.

2.1.5. Java

Obiektowy język programowania Java jest od wielu lat z powodzeniem wykorzystywany w wielu gałęziach przemysłu. Napisane w nim programy działają zarówno na urządzeniach mobilnych, w samochodach, telewizorach czy w wielkich centrach przetwarzania danych. Java zapewnia wysoką wydajność – wystarczającą dla większości zastosowań, a jednocześnie jest językiem niezbyt skomplikowanym i przystępnym. Przez wiele lat swojego istnienia zgromadziła wokół siebie dużą społeczność użytkowników oraz wielkie firmy, zainteresowane jej rozwojem. Najnowsze technologie najczęściej posiadają swoje implementacje wpasowujące się w ekosystem platformy Java. Stąd wybór tego języka programowania jako podstawy systemy klasy HIS dla sieci szpitali wydaje się być uzasadniony.

2.1.6. Budowanie numerów wersji

Zaawansowane systemy informatyczne składają się z wielu komponentów, z których każdy posiada zależności do takich czy innych bibliotek dostawców trzecich. Każdy z tych kawałków oprogramowania posiada swoją wersję, która powinna korespondować z oferowanym API (ang. *Application Programming Interface*) i sposobem jego implementacji. Wprowadzając zmiany czy poprawiając usterki, warto zmieniać numer wersji według określonych zasad, co pozwala uniknąć wielu błędów, np. wynikających z niekompatybilnych wstecz zmian w API. Wersjonowanie semantyczne[2] (ang. *semantic versioning*) jest szeroko stoso-

[2] Semantic Versioning: http://semver.org/

wanym standardem budowania numerów wersji komponentów czy całych systemów. Jego podstawą jest trójczłonowy numer wersji w postaci:

MAJOR.MINOR.PATCH

Zmiana na każdym z członów oznacza inny typ zmian. I tak:

- zmiana elementu MAJOR oznacza wprowadzenie niekompatybilnych wstecz zmian w API,
- zmiana elementu MINOR oznacza dodanie funkcjonalności z zachowaniem wstecznej kompatybilności,
- zmiana elementu PATCH oznacza zmiany naprawiające błędy, które w żaden sposób nie wpływają na publiczne API.

Ten sposób budowania numerów wersji jest zalecanym podejściem dla złożonych systemów informatycznych, takich jak system klasy HIS dla sieci szpitali.

2.2. Technologie

W poprzednich rozdziałach opisano podstawowe założenia architektury systemu klasy HIS dla sieci szpitali. Mają one charakter ogólny i uniwersalny, tj. znajdują zastosowanie w prawie każdym z jego komponentów. W większości są to obostrzenia wyznaczające granice, których nie powinno się przekraczać implementując poszczególne funkcje aplikacji. Odpowiednio dobrane technologie i narzędzia ułatwiają przestrzeganie tych zasad i usprawniają proces wytwarzania oprogramowania w przyjętej architekturze. W dalszej części rozdziału zostaną omówione najważniejsze technologie, które mogą zostać wykorzystane do budowy systemu klasy HIS dla sieci szpitali.

2.2.1. Spring Framework

Koncepcja architektury systemu klasy HIS dla sieci szpitali zakłada wykorzystanie języka Java jako podstawowego języka programowania, używanego do wytwarzania poszczególnych składników systemu (przynajmniej w warstwie przetwarzania). Java oferuje wiele technologii opisanych za pomocą specyfikacji, których wzorcowe implementacje dostarcza platforma Java Enterprise Edition. Wspomagają one programistę w rozwiązywaniu wielu zagadnień technicznych, napotykanych przy wytwarzaniu złożonych aplikacji (np. dostęp do bazy danych, komunikacja poprzez wiadomości itp.). Niestety, często to wsparcie jest niewystarczające, a narzucony model programowania niepotrzebnie komplikuje kod źródłowy.

Te problemy z powodzeniem adresuje Spring Framework[3]. Jest to zestaw bibliotek oferujący wszechstronny szkielet (ang. *framework*) tworzenia i konfigurowania złożonych aplika-

[3] Spring Framework Homepage: http://projects.spring.io/spring-framework

cji uruchamianych na platformie Java. Głównym jego zadaniem jest wyręczenie programisty w wielokrotnym pisaniu tych samych fragmentów kodu, które nie są bezpośrednio związane z funkcjonalnościami systemu (np. zamykanie połączeń do bazy danych). Spring oferuje wsparcie w wielu obszarach: począwszy od dostępu do bazy danych poprzez programowanie aspektowe, obsługę wiadomości JMS, przetwarzanie wsadowe, aż po tworzenie usług sieciowych w architekturze REST itp. Jedną z największych jego zalet jest mała „inwazyjność", tj. kod związany z modelem programowania oferowanym przez Spring, wydzielony jest do osobnych klas i plików konfiguracyjnych (w formacie XML), natomiast logika działania aplikacji zawarta jest w klasach typu POJO (ang. Plain Old Java Objects). Umożliwia to potencjalną migrację do innego rozwiązania bez konieczności ingerencji w kod źródłowy znajdujący się w klasach typu POJO.

Spring Framework poprzez swoją dojrzałość oraz szeroki wachlarz użytecznych komponentów i bibliotek może być z powodzeniem wykorzystany w tworzeniu systemu klasy HIS dla sieci szpitali.

2.2.2. Spring Boot

Jednym z podstawowych założeń proponowanej architektury systemu HIS dla sieci szpitali jest odejście od monolitycznej budowy aplikacji i migracja w stronę architektury opartej na mikrousługach. Takie podejście zapewnia większą elastyczność i ułatwia skalowanie całego systemu: w momencie gdy obciążenie niebezpiecznie rośnie, można powołać do życia dodatkowe instancje tych mikrousług, które są najbardziej wykorzystywane. Powodzenie takiej strategii wymaga ograniczenia do minimum liczby kroków wymaganych do uruchomienia mikrousługi i czasu, po jakim jest ona w pełni gotowa do realizacji swoich zadań. Osiągnięcie tych celów przy użyciu obecnie wykorzystywanych w wielu systemach serwerów aplikacji [17] jest niemożliwe z kilku powodów. Po pierwsze, zużywają one zbyt dużo zasobów (często na usługi, które nie są wykorzystywane w aplikacji), przez co uruchomienie ich w wielu instancjach jest nieefektywne/nieuzasadnione. Po drugie, proces instalacji aplikacji jest często skomplikowany i nie daje się w pełni zautomatyzować. Po trzecie, czas startu samego serwera jest często dłuższy niż czas uruchomienia samej aplikacji.

Rozwiązanie tych problemów oferuje technologia Spring Boot[4]. Wspiera ona tworzenie samodzielnych (ang. *stand-alone*) aplikacji opartych na Spring Frameworku. Aplikacja stworzona w tej technologii to pojedyncze archiwum JAR, które zawiera w sobie wszystkie wymagane przez siebie komponenty. Można ją wystartować najprostszą komendą służącą do uruchamiania aplikacji Java:

```
$ java -jar spring-boot-app.jar
```

[4] Spring Boot: http://projects.spring.io/spring-boot

Aplikacje Spring Boot mogą zawierać w sobie kontener HTTP implementujący specyfikację Java Servlet API, przez co w prosty i wydajny sposób mogą oferować swoje usługi innym klientom poprzez protokoły bazujące na HTTP (REST, SOAP WebServices, WebSockets itp.).

Każda aplikacja wymaga parametrów konfiguracyjnych określających kontekst jej działania. Technologia Spring Boot oferuje kilka sposobów na przekazanie ich do aplikacji, są to:

- pliki tekstowe (w formacie Java Properties lub YAML),
- parametry wiersza poleceń,
- zmienne systemowe.

Ze względu na możliwość uruchomienia systemu w chmurze lub środowisku wykorzystującym wirtualizację, zalecanym sposobem są zmienne systemowe.

W systemie informatycznym o architekturze opartej na mikrousługach, łatwość ich uruchamiania i konfigurowania jest sprawą kluczową. Technologia Spring Boot upraszcza każdy z tych procesów, przez co dobrze sprawdza się w takim zastosowaniu. Naturalna integracja z komponentami platformy Spring jest dodatkowym argumentem przemawiającym za wykorzystaniem tego rozwiązania do budowy systemu HIS dla sieci szpitali.

2.2.3. Apache CXF

Jednym z podstawowych założeń architektury systemu klasy HIS dla sieci szpitali jest zgodność z paradygmatami SOA. Wymierne korzyści z takiego podejścia do tworzenia oprogramowania osiągane są wtedy, gdy usługi aplikacji dostępne są w wielu protokołach z wykorzystaniem różnych mechanizmów komunikacji. Sprostanie temu wymaganiu wymusza dostosowanie kodu źródłowego aplikacji do często bardzo odmiennych modeli programowania, co niesie ze sobą ryzyko błędów i zwiększa koszty jego utrzymania i modyfikacji. W rozwiązaniu tego typu problemów pomaga biblioteka Apache CXF[5], która dostarcza narzędzia, pozwalające na odseparowanie kodu odpowiedzialnego za obsługę danego protokołu od klas realizujących logikę przetwarzania. Dzięki temu programista tworzy swój kod w modelu POJO, a biblioteka dba o udostępnienie jego funkcjonalności w żądany sposób – np. jako SOAP Web Services dostępne poprzez protokół HTTP.

2.2.4. Hystrix

Systemy informatyczne zbudowane w architekturze opartej na mikrousługach z definicji są silnie rozproszone. Poszczególne komponenty komunikują się ze sobą, wykorzystując różne mechanizmy sieciowe i protokoły, co zwiększa prawdopodobieństwo wystąpienia błędów i przerw w działaniu niektórych funkcji systemu. Aby je zminimalizować, już na etapie projektowania aplikacji trzeba wziąć pod uwagę ewentualne opóźnienia wynikające z transmisji

danych po sieci, zmianę jej konfiguracji, problemy bezpieczeństwa itp. (szczegółowy opis zagadnień związanych z tym tematem znajduje się w [9]). Dobrze zaprojektowany system rozproszony jest odporny na awarie różnego typu i błąd w jednym miejscu nie powinien rzutować na działanie całości oprogramowania.

Biblioteka Hystrix[6] wspomaga projektanta/programistę w przygotowaniu aplikacji (w szczególności mikrousługi) do obsługi błędów związanych z dostępem do zewnętrznych zasobów, które w rozproszonym środowisku prędzej czy później wystąpią. Dostarczane przez nią rozwiązania pozwalają na:

- kontrolę opóźnień wynikających z dostępu przez sieć do usług zależnych (innych mikrousług w ramach systemu lub usług zewnętrznych, np. eWUŚ),
- izolację błędów spowodowanych np. zbyt dużym obciążeniem i uniknięcie ich propagacji na inne składniki systemu,
- zwracanie „awaryjnych" (ang. *fallback*) odpowiedzi, jeżeli usługa nie może odpowiedzieć klientowi w standardowy sposób,
- monitorowanie stanu aplikacji.

Zastosowanie biblioteki Hystrix ułatwia wdrożenie spójnego mechanizmu prewencji i obsługi błędów przy odwoływaniu się do zewnętrznych usług/zasobów. Zaawansowane możliwości konfiguracji i śledzenia stanu systemu na bieżąco (ang. real-time), pozwalają dostosować to rozwiązanie do bardzo specyficznych potrzeb. Hystrix może być wykorzystywany nie tylko w systemach opartych na mikrousługach, ale także w klasycznych aplikacjach Java, które korzystają z usług sieciowych.

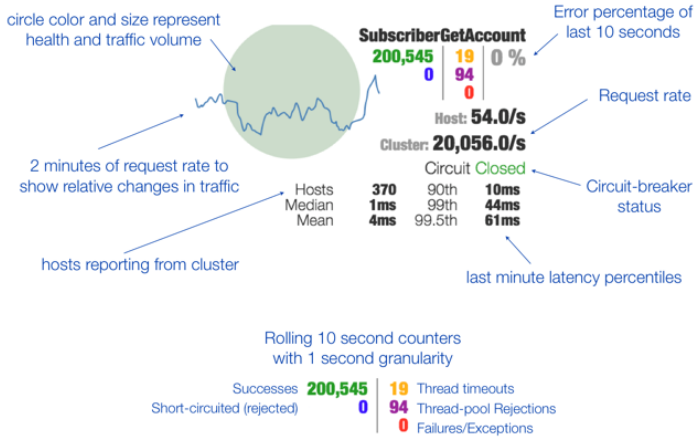
2.2.5. Narzędzia do monitorowania

W rozproszonym systemie informatycznym prawdopodobieństwo awarii jest znacznie większe niż w klasycznej aplikacji działającej w ramach jednego procesu. Co więcej, zlokalizowanie źródła problemu i jego diagnostyka są także nieporównywalnie trudniejsze. Narzędzia, takie jak wspomniana wcześniej biblioteka Hystrix, zapobiegają „roznoszeniu" się błędu „po" całym systemie, jednak ich funkcjonalność musi zostać uzupełniona o mechanizmy monitorowania i ostrzegania o wystąpieniu problemu. Tylko wtedy administrator ma szansę na szybką reakcję i zachowanie ciągłości pracy całego rozwiązania.

Do biblioteki Hystrix można dołączyć dodatkowe komponenty[7], które sprawiają, że na bieżąco można śledzić stan działania systemu. Rysunek 6 przedstawia przykładowy diagram ilustrujący dostępne wskaźniki dla jednego wywołania zdalnej usługi. Przejrzysty układ gra-

[5] Apache CXF: http://cxf.apache.org
[6] Hystrix GitHub Homepage: https://github.com/Netflix/Hystrix
[7] Hystrix Dashboard: https://github.com/Netflix/Hystrix/wiki/Dashboard

ficzny i intuicyjne kolory pozwalają szybko zorientować się, czy dana usługa pracuje normalnie, czy ma miejsce jakaś sytuacja wyjątkowa.



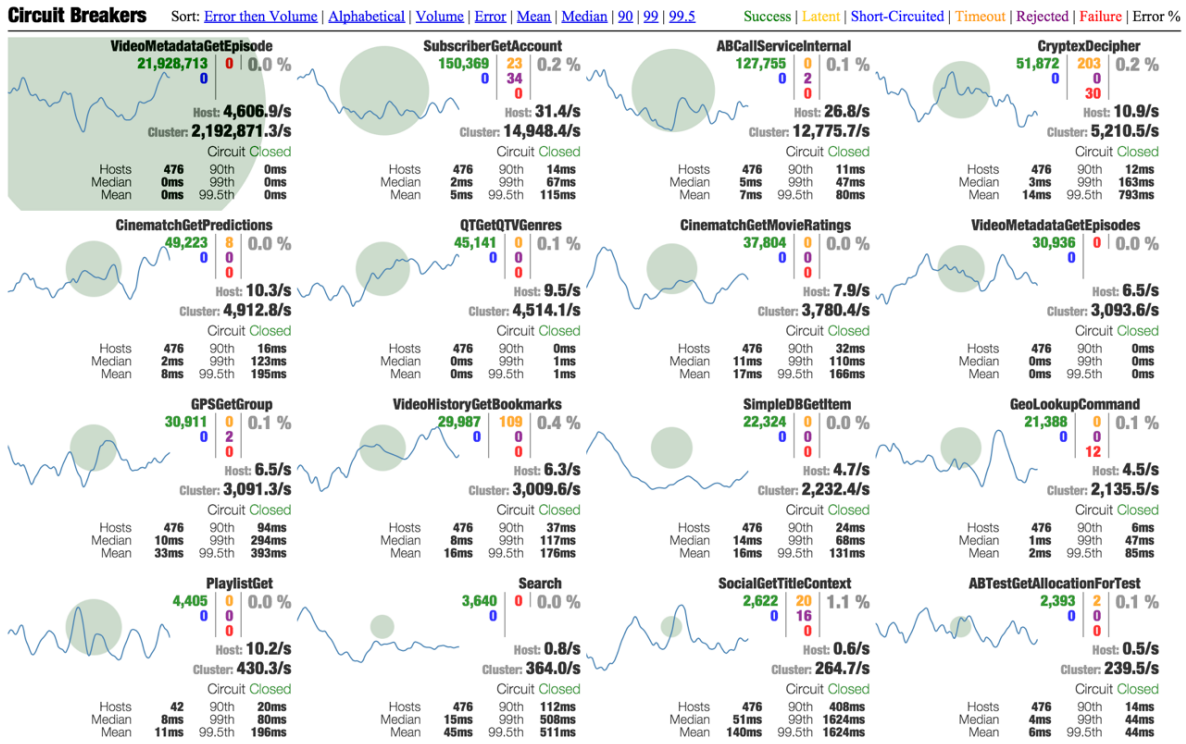
Rys. 3. Przykładowy diagram ze wskaźnikami działania wywołania zdalnej usługi[8]
Fig. 3. An example screen with gauges of remote service call

Jak wspomniano wcześniej, przedstawiony diagram wizualizuje stan działania tylko jednego odwołania do zdalnego zasobu, natomiast konkretna mikrousługa może komunikować się z wieloma takimi usługami/zasobami. Administrator systemu zbudowanego z mikrousług potrzebuje więc narzędzia, które pokaże stan działania całości systemu. Takim narzędziem jest Turbine[9]. Agreguje ono ww. diagramy w postaci jednej tablicy (ang. *dashboard*), która daje pogląd na działanie wszystkich składników aplikacji. Rysunek 4 pokazuje zrzut ekranu z przykładową tablicą.

Wspomniane wyżej narzędzia służą do wizualizacji stanu działania fragmentów oprogramowania wykorzystujących bibliotekę Hystrix. Ma ona wbudowane mechanizmy zbierające metryki użycia, które później są podstawą do generowania wykresów. Wytwarzając złożony system informatyczny, warto wprowadzić podobne rozwiązania. Do zbierania wskaźników działania aplikacji można wykorzystać komponent Metrics[10]. Pozwala on tworzyć cztery podstawowe typy mierników:

- wskaźniki (ang. gauges),
- liczniki (ang. counters),
- histogramy (ang. histograms),
- mierniki czasu (ang. timers).

[8] Źródło: https://github.com/Netflix/Hystrix/wiki/Dashboard
[9] Turbine Wiki: https://github.com/Netflix/Turbine/wiki
[10] Metrics: https://github.com/dropwizard/metrics



Rys. 4. Przykład tablicy monitorującej działania systemu w narzędziu Turbine[11]
Fig. 4. An example of dashboard of system monitoring in Turbine tool

Pierwszy z nich (gauges) służy do śledzenia pojedynczej wartości, np. długości kolejki zadań czy liczby podłączonych klientów itp. Drugi pozwala monitorować wartość całkowitą. Dzięki histogramom można obserwować, jak w ujęciu statystycznym rozkładają się wartości przyjmowane przez dany wskaźnik, np. rozmiar odpowiedzi wysyłanej do aplikacji mobilnej itp. Mierniki czasu pozwalają z kolei na gromadzenie informacji o czasie wykonania danego fragmentu kodu.

Zgromadzone wartości wskaźników trzymane są w pamięci operacyjnej maszyny, na której oprogramowanie zostało uruchomione. Biblioteka Metrics oferuje dodatkowe komponenty, z których pomocą można je okresowo utrwalać. W najprostszym przypadku można je zapisać do pliku w formacie CSV lub w pliku dziennika (ang. *log*). W bardziej zaawansowanych rozwiązaniach, zgromadzone metryki przesyła się do dedykowanego oprogramowania, które w optymalny sposób składuje dane i przetwarza je pod kątem późniejszej wizualizacji. Metrics współpracuje z narzędziami Graphite[12] i Ganglia[13]. Oba są stabilne i posiadają rozbudowaną funkcjonalność, jednak Graphite może być użyty w połączeniu z zaawansowanym narzędziem do wizualizacji – Grafana[14]. Służy ono do budowania interaktywnych tablic (dashboard), prezentujących wybrane wskaźniki, na jednym lub wielu wykresach. Większość

[11] Źródło: https://github.com/Netflix/Hystrix/wiki/images/dashboard-example-640.png
[12] Graphite Homepage: http://graphite.wikidot.com
[13] Ganglia Hompage: http://ganglia.sourceforge.net
[14] Grafana Homepage: http://grafana.org

obiektów wizualnych (wykresy, układ elementów, kolory) można dostosować do swoich potrzeb, aby szybko ocenić stan działania systemu.

Przedstawione w tym rozdziale narzędzia pozwalają na zbieranie, przechowywanie i wizualizację wskaźników stanu działania systemu. Dzięki nim administrator może nie tylko na bieżąco śledzić sytuację, ale także sprawdzić, co mogło być przyczyną awarii sprzed kilku dni. Warunkiem koniecznym jest oczywiście poprawne zaprojektowanie i zaimplementowanie wskaźników aplikacji już na wczesnym etapie tworzenia systemu. Gdy system działa już w środowisku klienta, zalecane jest także okresowe sprawdzanie przydatności poszczególnych miar i ewentualne ich usuwanie/zmiana lub dodanie zupełnie nowych.

2.3. Proces wytwórczy

Jednym z kluczowych czynników wpływających na jakość oprogramowania jest sam proces wytwarzania. Od jego organizacji w dużej mierze zależy to, czy przyjęte założenia architektoniczne są przestrzegane. Niewłaściwie dobrane praktyki i narzędzia, zły podział zadań i odpowiedzialności może sprawić, że nawet najlepsza koncepcja architektury realizowana przy użyciu najbardziej dopasowanych do niej technologii nie będzie mieć szans na realizację, a końcowy produkt okaże się mało funkcjonalny i niezgodny z oczekiwaniami klienta. Dlatego bardzo ważne jest dostosowanie procesu wytwórczego do przyjętej architektury oraz dobranie takich praktyk i narzędzi, które współgrają z wykorzystywanymi technologiami.

Podstawowym artefaktem procesu tworzenia oprogramowania jest kod źródłowy. To z niego budowane są poszczególne komponenty, z których składa się cały system, a więc powinno się mu poświęcić szczególną uwagę. Istotny jest dobór odpowiednich narzędzi, za pomocą których programista tworzy kolejne linie kodu, takich jak np. system kontroli wersji [12]. Aplikacja budowana w architekturze opartej na mikrousługach składa się z wielu niezależnych modułów (mikrousług), z których każdy może mieć kilka równolegle rozwijanych wersji, które są zainstalowane i działają u różnych klientów. Jednocześnie zupełnie nowe mikrousługi są dodawane bardzo rzadko (z wyjątkiem początkowej fazy projektu). Narzędzie do kontroli wersji powinno dopasować się do tego modelu, umożliwiając tworzenie wielu niewielkich repozytoriów oraz łatwe wydzielanie niezależnie rozwijanych gałęzi kodu. Te wymagania spełniają rozproszone systemy kontroli wersji [12], takie jak np. Git[15]. Niewątpliwą zaletą tego narzędzia jest brak sztywnego sposobu pracy (ang. *workflow*) z kodem źródłowym, przez co możliwe jest wypracowanie własnego, optymalnego podejścia [13]. Mikrousługi realizują zazwyczaj niewielki, jasno określony fragment funkcjonalności całego sys-

[15] Git Homepage: http://git-scm.com

temu, dlatego powinny być tworzone przez stosunkowo mały zespół ludzi[16]. Używając narzędzia Git, można zorganizować pracę zespołu w ten sposób, że programiści wprowadzają zmiany we własnych repozytoriach, następnie wysyłają je do akceptacji projektanta/architekta, i dopiero on wprowadza je do „oficjalnego" repozytorium danej mikrousługi. Ułatwia to także wprowadzenie praktyki przeglądów kodu (ang. code review), dzięki której łatwiej utrzymać określone standardy pisania kodu (formatowanie, nazewnictwo itp.) i stale podnosić poziom wiedzy poszczególnych członków zespołu [18].

Jak wspomniano wcześniej, duża zmienność jest immanentną cechą mikrousług, dlatego bardzo ważne jest wdrożenie w procesie wytwórczym mechanizmów, które bardzo szybko są w stanie wychwycić pojawiające się błędy. W ostatnich latach dużą popularność zyskało podejście tzw. ciągłej integracji (ang. *continuous integration*) [14]. Zakłada ono, że po każdej zatwierdzonej w repozytorium zmianie w kodzie źródłowym, uruchamiany jest proces budowy i automatycznego testowania aplikacji. Dzięki temu możliwe jest prawie natychmiastowe wychwycenie wprowadzonego błędu. Wymaga to jednak zaangażowania zespołu w przygotowanie i utrzymywanie zestawu testów jednostkowych i integracyjnych oraz poświęcenie czasu na automatyzację całego procesu. Przydatne mogą się okazać specjalizowane narzędzia, tzw. serwery budowy (ang. build servers) [15].

Systemy zbudowane w architekturze opartej na mikrousługach pozwalają niewielkim kosztem aktualizować wybrane fragmenty oprogramowania (bez konieczności zatrzymywania całego rozwiązania), dlatego przy ich tworzeniu szczególnie dobrze sprawdza się rozszerzenie techniki ciągłej integracji o elementy związane z uruchomieniem mikrousługi w środowisku produkcyjnym (np. w centrum przetwarzania). Koncepcja ciągłego dostarczania (ang. *continuous delivery*) [15] pozwala szybciej wdrażać nowe funkcje, reagować na zgłoszone błędy czy dostosowywać się do czynników zewnętrznych (np. zmiany w prawie). Zaproponowana architektura systemu HIS zakłada tworzenie poszczególnych mikrousług w technologii Spring Boot, która sprawia, że ich instalowanie, konfigurowanie i uruchamianie w docelowym środowisku daje się w pełni zautomatyzować, przez co nie powinno być znaczących problemów z adaptacją tego podejścia. Efektywne wprowadzenie w życie zasad ciągłego dostarczania wymaga także odpowiedniej struktury zespołu produkcyjnego. Powinien się on składać z ludzi, którzy posiadają szeroki wachlarz umiejętności: począwszy od projektowania/tworzenia kodu źródłowego poprzez testowanie aż do budowania i konfigurowania środowisk testowych i produkcyjnych. Niewielki rozmiar zespołu sprzyja wymianie wiedzy i podwyższaniu umiejętności poszczególnych osób, a także wypracowaniu wspólnej kultury pracy [16].

[16] The Science Behind Why Jeff Bezos's Two-Pizza Team Rule Works: http://blog.idonethis.com/two-pizza-team

3. Podsumowanie

Dynamiczny rozwój sieci Internet i technologii z nią związanych wpływa na przedsiębiorstwa w wielu gałęziach przemysłu. Jednostki ochrony zdrowia nie stanowią tutaj wyjątku. Każdego dnia muszą dostosowywać realizowane przez siebie procesy biznesowe do zmieniającego się otoczenia. Powoduje to, że posiadane przez nie oprogramowanie musi być stale udoskonalane i zdolne do komunikowania się ze światem zewnętrznym. Powszechny dostęp do szerokopasmowego Internetu oraz presja na obniżanie kosztów działalności szpitali są naturalnymi czynnikami przyspieszającymi ich konsolidację. Powstające sieci szpitali oczekują od dostawców oprogramowania produktów, które pozwolą osiągnąć dodatkowe korzyści ze współdziałania. Te uwarunkowania wymuszają stosowanie nowoczesnych architektur oprogramowania, które działają na podobnych zasadach jak globalna sieć i szeroko wykorzystują technologie na niej bazujące. W artykule przedstawiono koncepcję takiej właśnie architektury, dedykowanej dla systemu klasy HIS, przeznaczonego dla sieci jednostek ochrony zdrowia. Jej głównym elementem są mikrousługi, czyli autonomiczne komponenty realizujące ściśle określoną funkcjonalność i komunikujące się z innymi składnikami systemu poprzez asynchroniczne komunikaty i otwarte protokoły sieciowe bazujące na HTTP. Mikrousługi oferują swoje API różnego rodzaju aplikacjom klienckim (np. typu desktop, urządzeniom mobilnym itp.). W pracy zaproponowano także wiele zasad tworzenia poszczególnych składników systemu tak, aby całość rozwiązania sprostała wymaganiom co do wydajności, skalowalności i różnorodnych wariantów wdrożenia. Opisano także wybrane technologie i narzędzia, które ułatwiają stosowanie przyjętych założeń w praktyce. Ich użycie umożliwia także niezależne testowanie poszczególnych komponentów, automatyzację procesu budowy i uruchamiania. Zaproponowano także organizację wybranych fragmentów procesu wytwórczego, tak aby zapewnić wysoką jakość tworzonego oprogramowania, a także skrócić czas potrzebny na dostarczenie nowej wersji systemu.

Wskazówki i zalecenia zawarte w artykule mogą posłużyć do wytworzenia prototypowego oprogramowania klasy HIS dla sieci jednostek ochrony zdrowia. Po jego implementacji konieczne jest zweryfikowanie przyjętych założeń i ich ewentualna korekta.

BIBLIOGRAFIA

1. Fowler M.: Patterns of enterprise application architecture. Addison-Wesley Longman Publishing Co., Inc., 2002.
2. Brown S.: Software Architecture for Developers. Coding the Architecture, 2013.

3. Bosch J.: Software architecture: The next step. Software architecture. Springer, Berlin, Heidelberg 2004, s. 194÷199.
4. Dubray J.-J.: Composite Software Construction. Understanding SOA in the Context of a Programming Model. C4Media Inc, 2004.
5. Fowler M., Lewis J.: Microservices. http://martinfowler.com/articles/microservices.html.
6. Pattern: Microservices Architecture, http://microservices.io/patterns/microservices.html.
7. Pattern: Monolithic Architecture, http://microservices.io/patterns/monolithic.html.
8. Augustyn D. R.: Opis architektury wskazanego systemu HIS, będącego przedmiotem rozszerzenia pod kątem obsługi sieci jednostek, część a – Architektura biznesowa. Opracowanie wykonane w ramach zadania 1, podzadania 2 w ramach projektu NTHS, 2014.
9. Arnon R. G. O.: Fallacies of distributed computing explained. 2007.
10. Richardson L., Ruby S.: RESTful web services. O'Reilly Media, Inc., 2008.
11. The Reactive Manifesto, http://www.reactivemanifesto.org/.
12. Otte S.: Version Control Systems. Computer Systems and Telematics, Institute of Computer Science, Freie Universität, Berlin, Germany 2009.
13. Chacon S., Hamano J. C.: Pro git. Apress, Vol. 288, Berkeley, CA 2009.
14. Fowler M.: Continuous Integration, http://martinfowler.com/articles/ continuousIntegration.html
15. Humble J., Farley D.: Continuous delivery: reliable software releases through build, test, and deployment automation. Pearson Education, 2010.
16. Walls M.: Building a DevOps Culture. O'Reilly Media, Inc., 2013.
17. Pilewski B., Dżygadło B., Łepecka-Klusek C.: Szpitalne systemy informatyczne i czynniki warunkujące ich wdrażanie. Pielęgniarstwo XXI wieku, Nr 4 (37), 2011, s. 47÷53.
18. Bacchelli A., Bird C.: Expectations, outcomes, and challenges of modern code review. Proceedings of the 2013 International Conference on Software Engineering, IEEE Press, 2013, s. 712÷721.

Abstract

Nowadays, hospital information system (HIS) processes huge amount of data. There is still increasing number of concurrent users (practitioners, technical stuff or even patients) interacting with it. Each solution is also integrated with many other software systems, which

consume or produce some medical data, therefore information exchange must be done in a near real-time manner. The complexity of interaction between involved parties grows even bigger when several hospitals decide to federate. As this is an ongoing process in polish health care sector, HIS manufacturers face a big challenge to align their systems architecture with current requirements.

This article presents the concept of a modern software architecture for distributed HIS systems. It is based on a micro-services architecture and aim to help develop systems that are reliable, scalable and yet performant. Figure 1 presents overview of proposed architecture and outlines basic schema of component distribution. To achieve high performance and scalability, proposed architecture assumes asynchronous information exchange between components. Figure 2 shows example of such communication. The article also briefly describes some Java-based tools and technologies, that can be used to build micro-services without the burden of complex configuration and setup. One of the key challenges of distributed system architecture is monitoring. Tracking component's "health" requires enabling them to produce some metrics which then can be analyzed and visualized. In proposed architecture we show how popular open source libraries can be used to meet this requirement. This article also emphasis the meaning of well organized software development process, which should be aligned with proposed architecture in areas such as source code management and team size.

Design principles, technologies, tools and general guidelines described in this article should help to develop distributed HIS system which can be integrated with other applications and easily scaled to handle growing number of concurrent users.

Adresy

Łukasz WARCHAŁ: Asseco Poland S.A., ul. Chorzowska 50, 44-100 Gliwice, Poland, lukasz.warchal@asseco.pl.
Dariusz R. AUGUSTYN: Politechnika Śląska, Instytut Informatyki, ul. Akademicka 16, 44-100 Gliwice, Poland, daugustyn@polsl.pl.
Łukasz WYCIŚLIK: Politechnika Śląska, Instytut Informatyki, ul. Akademicka 16, 44-100 Gliwice, Poland, lwycislik@polsl.pl.