Interakcja Użytkownik – Agentowy System Wspomagania Podróży

Marcin PAPRZYCKI, Paweł KACZMAREK, Maciej GAWINECKI, Zygmunt VETULANI

Streszczenie: Niniejszy artykuł opisuje, odpowiedzialny za interakcje z użytkownikiem, podsystem agentowego systemu wspomagania podróży. W tekście zaprezentowana jest ogólna problematyka interakcji pomiędzy agentem a nie-agentem (klientem), wymagania, jakie przed sobą postawiliśmy budując ten podsystem oraz szczegółowy opis proponowanego przez nas rozwiązania.

1. Wprowadzenie

W chwili obecnej potencjalny turysta z dostępem do Internetu ma do wyboru szereg systemów wspomagających planowanie podróży, takich jak Expedia, Orbitz czy Travelocity, których zadaniem powinno być skuteczne dostarczenie poszukiwanej informacji. Niestety systemy te dostarczają tylko informacje związane z rezerwacją serwisów takich jak przeloty, czy noclegi. Równocześnie „ogólne systemy wyszukiwawcze", np. Google, zalewają turystów mało istotnymi informacjami. Wielu autorów, w tym P. Maes [2] sugeruje, że odpowiedzią na zalew informacją będą inteligentni agenci programowi. Ponieważ system planowania podróży wydaje się być idealnym scenariuszem do użycia technologii agentowej, powstało już kilkanaście projektów w tej dziedzinie. Niestety, większość z tych przedsięwzięć miało bądź ograniczony rozmiar lub nigdy nie wszyły z wczesnej fazy projektowania [3]. Naszym celem jest stworzenie działającego agentowego systemu wspomagania podróży. Oznacza to, że system ten ma wykorzystać **istniejące** narzędzia i technologie. Jak się okazuje, już jedno z najprostszych wymagań, aby użytkownik kontaktował się z agentem osobistym nie jest proste do zrealizowania i jego realizacji jest poświęcona niniejsza praca.

W następnej Sekcji zaprezentujemy ogólny opis budowanego przez nas agentowego systemu wspomagania podróży. W Sekcjach 3 i 4 przedyskutujemy możliwych klientów systemu agentowego oraz problem komunikacji systemu agentowego z użytkownikiem. Proponowane rozwiązanie oraz ilustrację przepływu danych omówimy w Sekcjach 5 i 6. Natomiast w Sekcji 7 opiszemy szczegóły implementacyjne i zilustrujemy je na konkretnym przykładzie działania systemu.

2. Opis Agentowego Systemu Wspomagania Podróży

Proponowany system został zaprojektowany tak, aby nie tylko służyć rozległą pulą usług: rezerwowanie miejsca w hotelu czy planowanie wycieczki, ale aby również realizować funkcje informacyjne takie jak: dostarczenie programu pobliskiego kina lub zasugerowanie restauracji o określonym profilu. System ten zbudowany został przy pomocy technologii agentowej (wszystkie najważniejsze funkcje realizowane są przez agentów programowych) połączonej z innymi technologiami, takimi jak bazy danych, ontologie, etc. Centralną częścią systemu jest repozytorium, w

którym przechowywane są (przy pomocy technologii JENA [4]) ontologicznie ustrukturyzowane dane opisane w języku znacznikowym RDF [5].

Użytkownik łączy się z systemem za pomocą różnorodnych urządzeń wejścia-wyjścia takich jak PDA, telefon komórkowy bądź też tradycyjna przeglądarka internetowa i zapytuje o interesujące go dane. Odpowiedzią na zapytanie użytkownika jest informacja pozyskana z centralnego repozytorium, przefiltrowana pod kątem osobistych upodobań klienta. W przypadku zapotrzebowania na dodatkową wiedzę agenci mogą przeszukać Internet gromadząc potrzebne informacje. Prezentowany rezultat poszukiwań jest „poprawiany" w procesie interakcji użytkownika z systemem. Wszystkie szczegóły tych interakcji są przechowywane w bazie danych „zachowań" dla późniejszego odnajdowania różnego typu schematów [6].

Jednym z najważniejszych celów, jakie przed sobą postawiliśmy było udostępnienie funkcjonalności naszego systemu dla jak najszerszego grona użytkowników Internetu. Użytkownik w podróży często zaopatrzony jest w laptop i/lub PDA i/lub telefon komórkowy. Każde z tych urządzeń cechuje zbiór ograniczeń, który trzeba było wziąć pod uwagę projektując system. Przejdziemy teraz do opisu tychże ograniczeń i ich konsekwencji dla implementacji komunikacji użytkownik-system.

3. Klienci systemu

Chcąc zbudować, platformowo niezależny system, zaczęliśmy analizę wymagań od strony użytkownika (klienta, którego będzie on używał). Natychmiastowo zaobserwować można dużą różnorodność istniejących klientów, różnorodność stanowiącą jedno z głównych źródeł problemów interakcji klient-agent. Wprawdzie, do grona interaktywnych klientów zalicza się aplikacje oparte o Flash lub o Java Applets, musieliśmy jednakże założyć, że użytkownik systemu nie zamierza lub nie może (z powodów administracyjnych) załadować takich aplikacji. Zmusza nas to również do wyeliminowania „ciasteczek" jako opcji technologicznej (istnieją użytkownicy/systemy, które uniemożliwią wykorzystanie tej technologii). Z podobnych powodów niemożliwym jest przyjęcie bardzo powszechnego założenia pomiędzy naukowcami zajmującymi się agentami: że agent będzie mógł przejść z systemu do urządzenia końcowego (rozwiązanie takie wymagało by załadowania odpowiedniego oprogramowania). Skoro istnieją urządzenia, które nie będą (z powodów technologicznych, administracyjnych lub innych) dostępne dla agentów, musimy znaleźć rozwiązanie pozwalające nam uniknąć tego wymagania. Innymi słowy, musimy zaimplementować rozwiązanie, które będzie działało w przypadku urządzeń cechujących się tylko minimalnymi (i wspólnymi dla wszystkich) możliwościami. Analiza przeprowadzona w [7] wskazuje, że taki minimalny zbiór wymogów to:

- Urządzenie klienta, powinno być w stanie interpretować jeden z języków (HTML, WML, XHTML, itp.).
- Klient musi być w stanie nawiązać jednokierunkowe połączenie poprzez Internet używając protokołu HTTP.

4. Komunikacja z agentami systemu

W poprzedniej sekcji poczyniliśmy założenie, iż nasz system wspomagania podróży będzie miał strukturę typu przeglądarka – system agentowy. Należy tu podkreślić, że założenie to łamie jedną z podstawowych wytycznych programowania agentowego: agenci powinni być wszędzie! Użytkownik powinien mieć własnego agenta na pulpicie, który reprezentowałby go w interakcjach z systemem (agentowym). W rzeczywistości jednak, zaistnienie agenta wymaga odpowiedniego środowiska tj. platformy agentowej (agencji). Wprawdzie odrzuciliśmy ideę aplikacji ładowanej na urządzenie, ale nawet gdyby to było możliwe to nie rozwiązałoby to naszego problemu. Istnieje obecnie wiele „agencji", jak: Aglets, Grasshopper, Voyager, Concordia, jednakże są one niekompatybilne – agenci nie mogą migrować na obcą platformę. Dodatkowo, nie każda agencja pozwala agentom na zasiedlanie się na mobilnych urządzeniach.

Biorąc pod uwagę, iż agenci nie mogą migrować na urządzenia mobilne, pozostaje nam wspomaganie wymiany wiadomości pomiędzy użytkownikiem a agentem umieszczone po stronie serwera. Niestety, okazuje się, że standardy komunikacji opracowane przez FIPA [8] dotyczą komunikacji pomiędzy agentami (np. język ACL [9]), natomiast agenci posługiwać się mają specjalnym dialektem HTTP, który sprawia, że nie mogą porozumiewać się z nie-agentami bez instalacji dodatkowego oprogramowania (co byłoby sprzeczne z naszymi założeniami). Nie jesteśmy więc w stanie wymieniać wiadomości z agentem sami nie będąc agentem.

Ostatnim z rozważanych rozwiązań byłoby uczynienie agenta serwerem. Agent ten zawarłby w sobie wsparcie zarówno dla protokołu HTTP, jak i języków znacznikowych jak HTML, WML czy XML. Jednakże agent, którego lokacja jest stała i który jedynie odpowiada na żądania, nie byłby już w rzeczywistości agentem. Ponadto, złożoność i rozmiar takiego agenta sprawiłyby, że byłby on *de facto* serwerem.

Jest więc oczywistym, że żaden z wymienionych scenariuszy użycia agentów: agent w urządzeniu, agent komunikujący się z klientem czy agent-serwer, nie rozwiązuje problemu komunikacji pomiędzy agentem a nie-agentem.

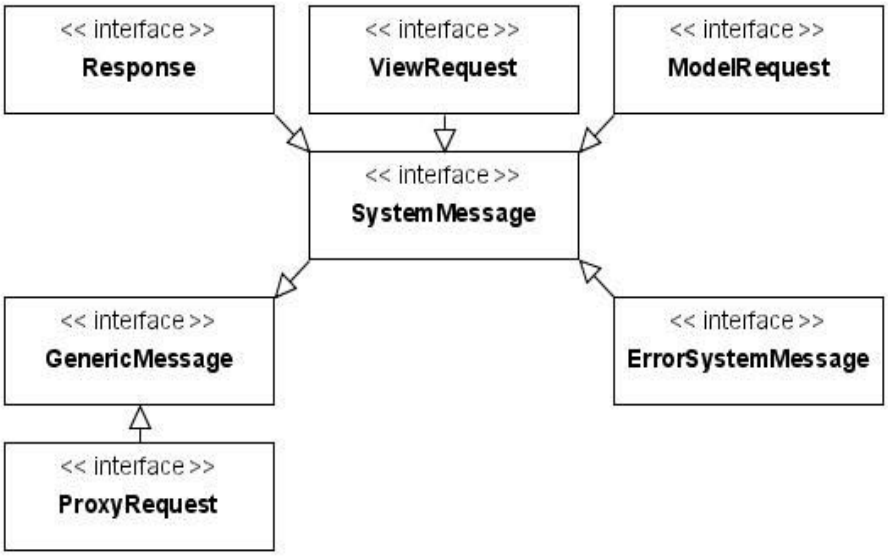
5. Proponowane rozwiązanie - wprowadzenie

W celu zbudowania kanału komunikacyjnego pomiędzy klientem (użytkownikiem) a (jego) *Agentem Personalnym* (*PA*) utworzona została warstwa pośrednicząca, która z jednej strony udostępnia zbiór komponentów nasłuchujących dla różnych typów mediów, z drugiej zaś strony komunikuje się z *PA* w zrozumiałym dla niego języku (ACL). Zadaniem tej warstwy jest, zatem przyjęcie żądania od klienta, stworzenie na tej podstawie żądania, które zrozumie *PA*, następnie odebranie wyników od *PA* i przekazanie ich do klienta w zrozumiałej dla niego formie.

Całość proponowanego rozwiązania można więc podzielić na dwie części:

a) zewnętrzną – która ma styczność z środowiskiem nie-agentowym,
b) wewnętrzną – która jest środowiskiem agentowym.

W celu zunifikowania obsługi żądań klientów pochodzących z różnych mediów, stworzyliśmy klasy kapsułkujące pojęcie żądania (*Request*) i odpowiedzi (*Response*); schemat rozwiązania przedstawiony jest na Rysunku 1.



Rysunek 1. Schemat rozwiązania.

Zapytania przychodzące z zewnątrz systemu agentowego to wiadomości typu *ProxyRequest*. Każda wiadomość w wewnętrznej części systemu musi implementować interfejs *SystemMessage*. Oto sygnatura interfejsu *SystemMessage*:

```
AID getSender();
String getID();
String getName();          // odziedziczone z GenericMessage
String getContent();       // odziedziczone z GenericMessage
```

Wszystkie najważniejsze informacje (parametry żądania czy instancje ontologii) w wewnętrznej części systemu są przesyłane jako zawartość wiadomości ACL. *Content* wiadomości ACL opowiada *Content SystemMessage*. Każda klasa implementująca interfejs dziedziczący z *SystemMessage* ma dodatkową metodę, która potrafi zinterpretować *Content* wiadomości. Dla przykładu *ModelRequest* ma metodę

```
Map getParams(),
```

która potrafi zamienić *Content* na mapę parametrów postaci „klucz” – „wartość”.

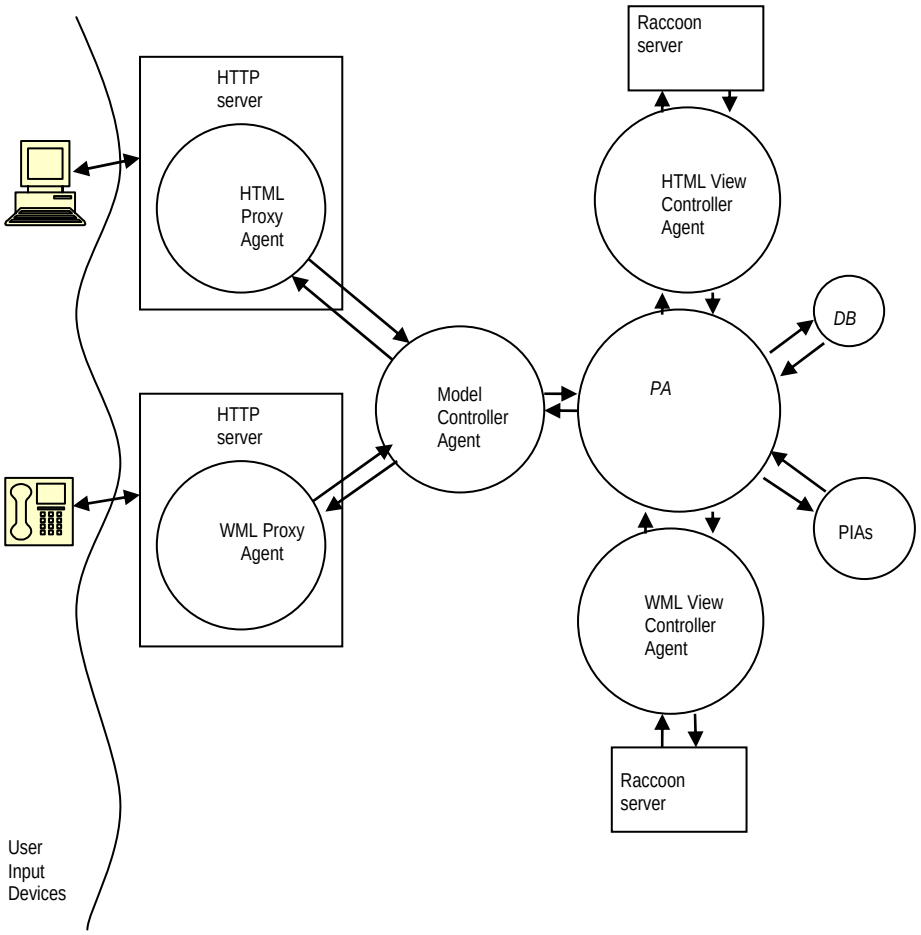
W celu ukrycia implementacji operacji na wiadomościach typu *SystemMessage* przed użytkownikiem powstała klasa abstrakcyjna, *SystemAgent*, która dziedziczy z klasy *Agent* platformy JADE. *SystemAgent* posiada metody takie jak:

sendSystemMessage (*SystemMessage msg*)
SystemMessage blockingReceiveSystemMessage ()
ModelRequest blockingReceiveModelRequest ()

(Ta ostania metoda powstała dzięki rozszerzeniu klasy *MessageTemplate* platformy JADE, która pozwalała na wybiórcze odbieranie przychodzących wiadomości.) Każdy agent wewnątrz systemu dziedziczy z *SystemAgent* wynosząc tym samym wymianę wiadomości wewnątrz systemu ma poziom abstrakcji *SystemMessage*.

6. Przepływ danych

Schemat przepływu danych w podsystemie komunikacji klient-agent zilustrowany został na Rysunku 2. Działanie podsystemu zaczyna się od *Medium Proxy Agenta* (*Medium_PrA*) osadzonego w serwerze HTTP. Jego zadaniem jest uruchomienie modułu nasłuchującego oraz obsługa nadchodzących żądań – jest on pośrednikiem pomiędzy „światem zewnętrznym" i „światem agentowym". Każdy *Medium Proxy Agent* transformuje wiadomości *ProxyRequest* na *SystemMessage*. Na rysunku 2 mamy dwóch takich agentów: agenta HTML i agenta WML.



Rysunek 2. Przepływ informacji w podsystemie komunikacji klient-agenci.

Wiadomość przesłana zostaje następnie do *PA*, który ponosi całkowitą odpowiedzialność za obsłużenie zapytania klienta. Pierwszym krokiem, który musi wykonać *PA* jest pobranie z „bazy danych JENA" informacji będących odpowiedzią na zapytanie użytkownika. W tym celu, wysyła on wiadomości *Model Request* do *Data Base Agen*t (*DBA*). *DBA* jest odpowiedzialny za interakcje z JENA, czyli za wykonanie zapytania na repozytorium danych w oparciu o otrzymane parametry użytkownika. W pierwszej kolejności *DBA* zamienia mapę parametrów „klucz" – „wartość" na zapytanie RDQL [10], następnie, wykonuje właściwe zapytanie w rezultacie otrzymując pewien zbiór trójek RDF. Otrzymany rezultat, przesłany zostaje (jako zawartość wiadomości ACL) następnie do *PA*, który przesyła wyniki do agentów zajmujących się personalizacją treści; ich opis zostanie tutaj pominięty [3]. Po otrzymaniu odpowiedzi, *PA* wykorzystuje profil użytkownika w celu przygotowania ostatecznej odpowiedzi, która przesłana musi zostać użytkownikowi. W tym celu, *PA* wysyła wiadomość *View Request* zawierającą rezultaty działania systemu do odpowiedniego *Medium View Controller Agenta* (np. *WML Controller Agent*), którego zadaniem jest przekształcenie informacji na postać zrozumiałą dla docelowego klienta. W tym celu wykorzystany zostaje serwer Raccoon [11]. Jest to aplikacja, w swej funkcjonalności przypominającą Cocoon fundacji Apache. Różnica polega na tym, iż Raccoon zamiast obsługiwać czysty XML, przygotuje do wyświetlenia dane RDF, przekształcając je na formę, która może być użyta w interesujących nas urządzeniach (HTML, WML, itp.).

7. Szczegóły implementacji, przykład

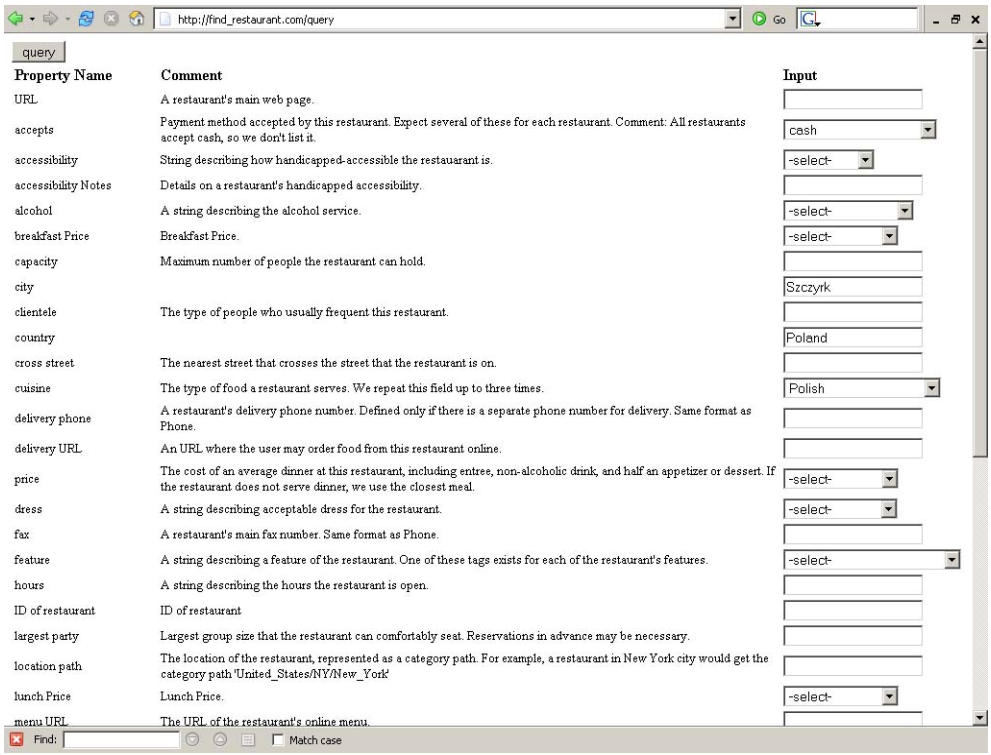
Dla ustalenia uwagi, skupmy się na opisie działania systemu na przykładzie tradycyjnej przeglądarki HTML – w przypadku, gdy klient chce znaleźć restaurację w Szczyrku. Załóżmy, że agenci *HTML Proxy Agent*, *HTML View Controller Agent*, *Model Controller Agent*, oraz *PA* zostali stworzeni a repozytorium JENA zawiera dane dotyczące restauracji w Polsce. Jednym z zadań *HTML Proxy Agent* jest udostępnienia modułu nasłuchującego. W tym celu, agent ten uruchamia miniaturowy serwer HTTP, który będzie nasłuchiwał pod ustalonym adresem i portem. W celu udostępnienia funkcjonalności *HTML Proxy Agenta*, serwerowi HTTP przekazywana jest instancja wewnętrznej klasy *HTMLProxyAgent*, która zawiera publiczną metodę

```
Response processRequest(ProxyRequest request).
```

Zadaniem tej metody jest zwrócenie żądanych informacji w gotowej do wyświetlenia formie, czyli w tym przypadku – HTML. Załóżmy dodatkowo, że formularz zapytujący został już wygenerowany, i klient wysłał wypełniony formularz do serwera (przykład formularza przestawiony został na Rysunku 3). Jak widać, parametry wyszukiwania są następujące:

```
Country = Poland
City = Szczyrk
```

```
Cousine = Polish
Accepts = cash
```



Rysunek 3. Formularz zapytania.

Serwer otrzymuje, więc odpowiedni querystring z zaszytymi parametrami zapytania oraz z nazwą akcji. Wątek obsługujący żądanie, na podstawie otrzymanego querystring'a tworzy instancje wiadomości *ProxyRequest* i wywołuje wyżej wymienioną metodę „*processRequest*". W ten sposób, w *HTMLProxyAgent* uruchamia się mechanizm obsługi tego żądania. Wątek serwera HTTP obsługujący żądanie zasypia, czekając na odpowiedź. Instancja *ProxyRequest* zostaje przekonwertowana na *ModelRequest* (jest już to wiadomość wewnętrzna, więc przydzielony jest jej unikalny ID) i wysłana do *Model Controller Agenta*. Agent ten, na podstawie nazwy użytkownika wybiera odpowiedniego *PA,* do którego posyła zapytanie klienta. *Agent Personalny* komunikuje się z *DBA*, przekazując mu parametry zapytania. *DBA* posługując się otrzymaną wiadomością generuję zapytanie RDQL, które wykorzystuje w interakcji z JENA. Zapytanie w naszym przypadku wygląda następująco:

```
SELECT ?res
 WHERE
(?res,
```

```
    <http://www.agentlab.com/schemas/Restaurant#cuisine>,
    <http://www.agentlab.com/schemas/CuisineCode#Polish>),
    (?res,
    <http://www.agentlab.com/schemas/Restaurant#city>,
    'Szczyrk'),
    (?res,
    <http://www.agentlab.com/schemas/Restaurant#accepts>,
    <http://www.agentlab.com/schemas/MeanOfPayment#Cash>),
    ?res,
    <http://www.agentlab.com/schemas/Restaurant#country>,
    'Poland')
```

W odpowiedzi otrzymujemy instancje ontologii:

```
<j.0:Restaurant
rdf:about="http://www.agentlab.com/db/Restaurant#Poland_FS_Szczyrk__Beskidek
1046490541">
       <j.0:cuisine
rdf:resource="http://www.agentlab.com/schemas/CuisineCode#Polish"/>
       <j.0:accepts
rdf:resource="http://www.agentlab.com/schemas/AcceptsCode#cash"/>
       <j.0:zip>43-370</j.0:zip>
       <j.0:country>Poland</j.0:country>
       <j.0:state>DS</j.0:state>
       <j.0:id>Poland/FS/Szczyrk/Beskidek,Restauracja1046490541</j.0:id>
       <j.0:hours>10-22 daily</j.0:hours>
       <j.0:streetAddress>ul. Górska 90</j.0:streetAddress>
       <j.0:title>Beskidek, Restauracja</j.0:title>
       <j.0:city>Szczyrk</j.0:city>
       <j.0:locationPath>Poland/LS/Szczyrk</j.0:locationPath>
       <j.0:phone>+48 (33) 817 86 26 </j.0:phone>
</j.0:Restaurant>
```

oraz

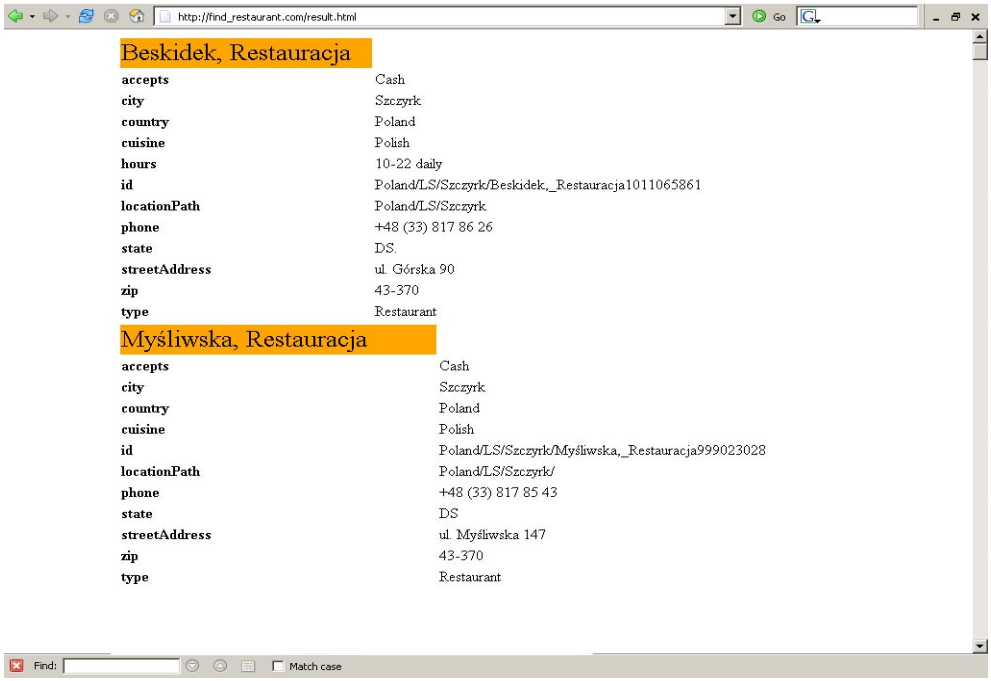
```
<j.0:Restaurant
rdf:about="http://www.agentlab.com/db/Restaurant#Poland_FS_Szczyrk__Mysliwsk
a999023028">
       <j.0:cuisine
rdf:resource="http://www.agentlab.com/schemas/CuisineCode#Polish"/>
       <j.0:accepts
rdf:resource="http://www.agentlab.com/schemas/AcceptsCode#cash"/>
       <j.0:zip>43-370</j.0:zip>
       <j.0:country>Poland</j.0:country>
       <j.0:state>DS</j.0:state>
       <j.0:id>Poland/FS/Szczyrk/Mysliwska,Restauracja999023028</j.0:id>
       <j.0:hours>10-22 daily</j.0:hours>
       <j.0:streetAddress>ul. Myśliwska 147</j.0:streetAddress>
       <j.0:title>Myśliwska, Restauracja</j.0:title>
       <j.0:city>Szczyrk</j.0:city>
       <j.0:locationPath>Poland/LS/Szczyrk</j.0:locationPath>
       <j.0:phone>+48 (33) 817 85 43</j.0:phone>
</j.0:Restaurant>
```

Pomijamy tutaj struktury personalizacji, które mogłyby dodać nowe lub usunąć istniejące restauracje, i zakładamy, że te dwie restauracje stanowią ostateczną odpowiedź. Odpowiedź ta przekazana zostaje (jako zawartość wiadomości ACL) do *HTML View Controller Agenta*. Agent ten wyznacza odpowiedniego

„pracownika” (wzorzec projektowy – Controller), który zajmie się przekształceniem wiadomości. Pracownik ten używa serwera Raccoon, aby przekształcić dane RDF na HTML. Dokładniej, instancje ontologii RDF/XML wydobyte z ViewRequest zapisywane są w pliku, o nazwie: „ID_zapytania”.rdf, w ścieżce wyszukiwań serwera Raccoon. Następnie „pracownik” przesyła do serwera Raccoon zapotrzebowanie:

```
http://raccoon_domena/„ID_zapytaniia”.rdf
```

W odpowiedzi otrzymuje instancje ontologii przedstawione w postaci HTML. Odpowiedź zwracana jest poprzez *HTML View Controller Agenta* i *PA* do *HTMLProxyAgent*'a. Ten, otrzymawszy odpowiedź (*Response*) zawiadamia o tym czekające wątki serwera HTTP. Na podstawie ID odpowiedź trafia „we właściwe ręce” i zostaje wyświetlona w formie przestawionej na Rysunku 4.



Rysunek 4. Odpowiedź systemu na zapytanie o restauracje w Szczyrku.

8. Uwagi końcowe

Niniejszy artykuł omawia zagadnienie interakcji agentów będących częścią Agentowego Systemu Wspomagania Podróży, z użytkownikami tegoż systemu (klientami). Pokazaliśmy w nim, że aktualnie istniejące technologie agentowe nie sprzyjają tworzeniu realistycznych aplikacji. Zaproponowaliśmy również rozwiązanie, które w sposób możliwie najprostszy pozwala nam obejść ograniczenia narzucone przez istniejące technologie i użycie agentów

programowych w realistycznym systemie. Poprzez zaprzęgnięcie rozpowszechnionym standardów takich jak HTTP czy języki znacznikowe zdołaliśmy ustanowić komunikację pomiędzy bardzo „cienkim" klientem a systemem składającym się z agentów. Zauważmy tutaj, iż zaproponowane rozwiązanie daje się w sposób naturalny rozszerzyć do sytuacji w której agenci będą mogli przejść przynajmniej do niektórych urządzeń wejścia-wyjścia. Wystarczy tylko uruchomić komunikację agent-w-urządzeniu – agent personalny, natomiast reszta systemu pozostaje bez zmian.

W chwili obecnej posiadamy zaimplementowany system działający z przeglądarką HTML oraz jesteśmy w trakcie implementacji interakcji z przeglądarką opartą o WML. Następnym krokiem będzie oprogramowanie agenta umiejscowionego w telefonie mogącym wykorzystywać oprogramowanie przygotowane w języku Java. O dalszym postępie naszych prac poinformujemy w najbliższym czasie.

Literatura:

1. Travel Support Project: http://www.agentlab.net/projects/e-Travel/
2. P. Maes, 1994, Agents that Reduce Work and Information Overload; Communications of the ACM, 37, 7, 31-40
3. R. Angryk, V. Galant, M. Gordon, M. Paprzycki, 2002, Travel Support System - an Agent-Based Framework, w: H. R. Arabnia, Y. Mun (ed.), *Proceedings of the International Conference on Internet Computing* (IC'02), CSREA Press, Las Vegas, NV, 719-725
4. http://www.hpl.hp.com/semweb/jena.htm
5. http://www.w3.org/RDF
6. V. Galant, M. Paprzycki, 2002, Information Personalization in an Internet Based Travel Support System, w: Abramowicz (ed.), *Proceedings of the BIS'2002 Conferenc*e, Poznań University of Economics Press, Poznań, Poland, 191-202
7. M. Gordon, M. Paprzycki, V. Galant, 2001, Agent-Client Interaction in a Web-based E-commerce System, w: D. Grigoras (ed.), *Proceedings of the International Symposium on Parallel and Distributed Computing*, University of Iaşi Press, Iaşi, Romania, 1-10
8. http://www.fipa.org/
9. http://www.fipa.org/specs/fipa00061/
10. http://www.w3.org/Submission/2004/SUBM-RDQL-20040109/
11. http://rx4**rdf**.liminalzone.org/Racoon

Maciej Gawiniecki, Paweł Kaczmarek, Zygmunt Vetulani
Wydział Matematyki i Informatyki, Uniwersytet im. A. Mickiewicza
ul. Umultowska 87, 61-614 Poznań

Marcin Paprzycki
Instytut Informatyki, SWPS, ul. Chodakowska 18/31, 03-815 Warszawa
e-mail: Marcin.Paprzycki@swps.edu.pl