STUDIA INFORMATICA 2003
Volume 24 Number 4 (56)

Katarzyna HARĘŻLAK
Politechnika Śląska, Instytut Informatyki

AKTUALIZACJA KOPII DANYCH Z WYKORZYSTANIEM AGENTÓW PRZENOŚNYCH

Streszczenie. Opracowanie definiuje pojęcie agenta przenośnego, charakteryzuje jego cechy i mechanizmy działania w systemie Aglets Software Development Kit (ASDK). Skoncentrowano się na zastosowaniu technologii agentów przenośnych do realizacji transakcji w rozproszonych bazach danych. Określono zadania dla agentów przenośnych wykorzystywanych do realizacji aktualizacji kopii danych.

Słowa kluczowe: replikacja danych, przenośni agenci

UPDATES OF REPLICATED DATA WITH USAGE OF MOBILE AGENTS

Summary. The paper presents a definition of the mobile agent, its features and mechanisms in the Aglets Software Development Kit system (ASDK). The author discusses the processing of transactions in distributed databases using the mobile agent technology. The functions of mobile agents in replicated data management are also considered in the paper.

Keywords: data replication, mobile agents

1. Wstęp

Badania dotyczące przenośnych agentów osiągnęły punkt, w którym powstało wiele rozwiązań dla fundamentalnych problemów związanych z realizacją przetwarzania rozproszonego. Zaimplementowane zostały aplikacje i modele obliczeń rozproszonych wykorzystujące tę technologię. Istnieją opinie, że tradycyjne implementacje działają szybciej, są prostsze, bezpieczniejsze i bardziej odporne na awarie. Jednak sukces nowych rozwiązań związany jest z faktem, że doskonale sprawdzają się tam, gdzie od aplikacji wymagana jest duża elastyczność.

Tworząc system oparty na wykorzystaniu agentów, należy wziąć pod uwagę następujące aspekty [9, 11]:

1. Budowana aplikacja powinna mieć łatwość dodawania agentów.
2. Agent musi umieć manipulować zdarzeniami pochodzącymi ze świata zewnętrznego lub od innych agentów i powiadamiać o tych zdarzeniach innych agentów.
3. Musi istnieć możliwość dodania do agenta domeny wiedzy oraz procedur wnioskowania.
4. Agent powinien posiadać zdolność uczenia się w celu przeprowadzania klasyfikacji wiedzy.
5. Wieloagentowa aplikacja musi być wspierana przez wykorzystanie protokołu wymiany komunikatów.
6. Agent musi być trwały. Jak zostanie stworzony, musi umieć zachować swój stan, by załadować go w późniejszym czasie.

Rozważania dotyczące zadań aplikacji wykorzystującej technologię agentów prowadzone są w kontekście zastosowania ich w systemie zarządzania rozproszoną bazą danych, w szczególności do realizacji transakcji rozproszonych. Specyfiką takich transakcji jest fakt, że dotyczą one informacji znajdujących się w różnych węzłach sieci. Zatem do zadań stawianych przed agentami należy zaliczyć także umiejętność przemieszczania się pomiędzy węzłami sieci komputerowej w celu realizacji powierzonych im zadań. Wśród nich może znaleźć się np. aktualizacja bądź odtwarzanie danych lub nadzorowanie i raportowanie stanu sieci. W niniejszym opracowaniu uwagę skupiono na systemie Aglets Software Development Kit (ASDK) jako na narzędziu, które pozwala zrealizować postawione cele.

2. Definicja przenośnego agenta w środowisku Javy

W środowisku Javy agent przenośny zwany jest agletem [1]. Nazwa ta pochodzi od połączenia dwóch określeń: agent oraz applet – kod programu ładowany z serwera przez przeglądarki WWW. Aglety Javy są obiektami, które mogą:

- przenosić się z jednego węzła sieci komputerowej do innego,
- w dowolnym momencie przerwać swoje działanie i rozpocząć je w tym samym punkcie w innym węźle,
- przenosić się ze swoim aktualnym stanem i kodem,
- wykorzystywać swój mechanizm zabezpieczeń w celu ochrony przed podejrzanymi agentami.

Podstawową funkcjonalność przenośnych agentów w systemie ASDK definiuje Aglet API. Abstrakcyjna klasa *Aglet* zawiera podstawowe metody wykorzystywane do kontroli

mobilności i cyklu życia agenta. Wszyscy przenośni agenci zdefiniowani w *Aglets* muszą rozszerzać tę abstrakcyjną klasę [10].

2.1. AgletContext

Klasa AgletContext dostarcza interfejs dla środowiska wykonawczego, w którym działa aglet. Każdy aglet musi działać wewnątrz jakiegoś kontekstu. Tworzenie nowego kontekstu może odbywać się wewnątrz aplikacji lub wewnątrz serwera agletów za pomocą:

public abstract AgletContext ***createAgletContext****(String name).*

Po utworzeniu nowego kontekstu, pracujący w nim Aglet może odwołać się do niego za pomocą metody:

public final AgletContext ***getAgletContext****().*

Uzyskana w ten sposób informacja może posłużyć do ustalenia adresu lokalnego węzła, liczby działających w tym kontekście agentów lub do stworzenia nowego agenta.

2.2. AgletProxy

Interfejs AgletProxy umożliwia definiowanie dla każdego agletu uchwytu, stanowiącego jego osłonę. Osłona ta zapewnia jednolity i bezpieczny sposób komunikowania się z agletem. Jest ona wykorzystywana do ochrony publicznych metod, które nie powinny być w bezpośredni sposób dostępne dla innych obiektów. Inną ważną przyczyną użycia AgletProxy jest uzyskanie przeźroczystości lokalizacji agenta.

Dostęp do osłony agletu odbywa się z wykorzystaniem następujących metod:

- *public static AgletProxy* ***getAgletProxies****(java.lang.String contextAddress)* – metoda ta zwraca informacje o wszystkich agletach rezydujących w danym kontekście,
- *public static AgletProxy* ***getAgletProxy****(AgletID id)* – metoda zwraca uchwyt do agleta wskazanego jej argumentem.

3. Obiekt typu Aglet i jego cykl życia

Agenta tworzy się na postawie definicji klasy com.ibm.aglet.Aglet przez wywołanie metody:

public static AgletProxy ***createAglet****(java.lang.String contextAddress,*
java.net.URL codebase, java.lang.String classname,
java.lang.Object init).

Otrzymuje on swój unikalny numer *AgletId* niezmienny podczas całego jego życia. Innym sposobem powołania do życia agletu jest stworzenie kopii istniejącego agenta metodą:

public final java.lang.Object ***clone****()*.

Powstały obiekt posiada taki sam stan jak obiekt oryginalny, ale przydziela mu się inny identyfikator *AgletId*.

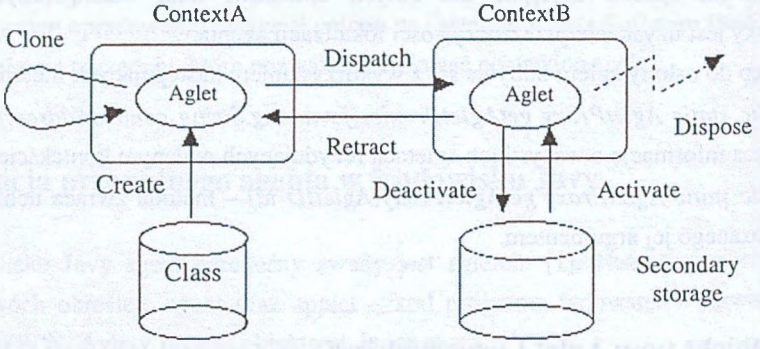
Podstawowe informacje o aglecie można uzyskać za pomocą metody:

public final AgletInfo ***getAgletInfo****()*.

Są to dane dotyczące bieżącego miejsca pobytu agleta, czasu jego przybycia, miejsca i czasu utworzenia.

Raz stworzony obiekt może być wysłany do, lub odwołany z innego węzła, dezaktywowany i później aktywowany. Operacje te mogą być realizowane z wykorzystaniem metod:

1) *public final void* ***dispatch****(java.net.URL destination)* – przeniesienie agenta z lokalnego węzła do miejsca przeznaczenia wskazanego jako argument metody,
2) *public abstract AgletProxy* ***retractAglet****(URL url, AgletID aid)* – przywołanie agenta do lokalnego węzła; parametry wskazują obecne miejsce pobytu i identyfikator agenta,
3) *public final void* ***deactivate****(long duration)* – umieszczenie agenta w specjalnym obszarze zwanym secondary storage,
4) *public abstract void* ***activate****()* – przywrócenie aktywności agletu,
5) *public final void* ***dispose****()* – zniszczenie i usunięcie agletu z jego bieżącego kontekstu.



Rys. 1. Cykl życia agletu
Fig. 1. The life cycle of aglet

Przykład:

```
URL destination = new URL ("atp://luna.iinf.polsl.gliwice.pl:1525/")
AgletContext ac = getAgletContext();
AgletProxy proxy = ac.createAglet(null, null, "Nowy aglet", null);
proxy = proxy.dispatch(destination); // metoda dispatch zwraca uchwyt wysyłanego aglata
proxy = ac.retractAglet(destination, proxy.getAgletID()); //pobranie agenta z powrotem, metoda retract
                                                          zwraca uchwyt pobieranego agenta
proxy.dispose();
```

4. Komunikacja agletów

Aglety komunikują się przez wymianę obiektów klasy Message: *public class Message extends Object implements Serializable.* Klasa ta udostępnia kilka konstruktorów pozwalających utworzyć wiadomość, nadając jej typ i opcjonalnie związać z nią wartość. Są to np.:

public Message(String kind);
public Message(String kind, int i);
public Message(String kind, double d);

Atrybuty wiadomości mogą być ustawiane i odczytywane odpowiednio za pomocą metod:

public void setArg(String name, Object a)
public Object getArg()

Przykład:

```
Message msg = new Message ("Operacja");
msg.setArg("SQL","select * from tabela1");
msg.setArg("nTr", 5);
msg.setArg("nOp", 1);
```

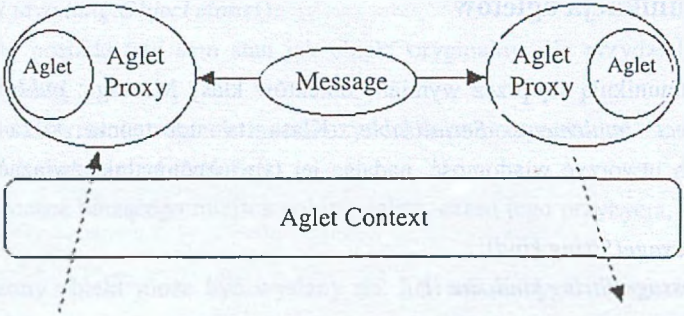
Obiekt typu wiadomość może być wysłany do innego agenta za pomocą następujących metod:

1) *public abstract Object sendMessage(Message msg)* – metoda synchroniczna, blokująca działanie agenta do czasu nadejścia odpowiedzi,
2) *public abstract FutureReply sendAsyncMessage(Message msg)* – metoda nie blokująca, nadejście odpowiedzi na wysłaną wiadomość sprawdzane jest asynchronicznie,
3) *public abstract void sendOnewayMessage(Message msg)* – metoda asynchroniczna, nie blokująca, wysyłająca wiadomość jednokierunkową.

Wiadomości rozróżnialne są na podstawie ich nazwy – atrybut *kind* – co daje możliwość dostosowania sposobu reakcji na ich nadejście. W tym celu należy zdefiniować własną procedurę obsługi wiadomości:

public boolean handleMessage(Message message)

W przesyłaniu wiadomości ważną rolę odgrywa interfejs AgletProxy. Obiekt Message dostarczany jest agletowi poprzez jego osłonę. Schemat przesyłu wiadomości ilustruje rys. 2.



Rys. 2. Schemat przesyłu wiadomości
Fig. 2. The message-passing schema

Przykład:

```
URL destination = new URL ("atp://luna.iinf.polsl.gliwice.pl:1525/")
AgletContext ac = getAgletContext();
AgletProxy proxy = ac.createAglet(null, null, "Nowy aglet", null);
proxy = proxy.dispatch(destination);
proxy.sendMessage("gdzie jesteś ?");
```

Jeżeli aglet wysyłający wiadomość jest zainteresowany odpowiedzią, wtedy postać wywołania metody jest następująca:

```
String reply = (String) proxy.sendMessage(new Message("gdzie jesteś ?"))
```

5. Rozszerzenie klasy Aglet

Podstawowa funkcjonalność klasy Aglet może zostać rozszerzona poprzez implementację następujących metod:

1) *Void Aglet.onCreation (Object init)* – wywoływanej tylko raz w całym cyklu życia agletu, zaraz po jego utworzeniu,
2) *Void Aglet.onDisposing()* – wywoływanej tuż po metodzie *dispose()*; aglet powinien w tym miejscu zwolnić wszystkie przydzielone mu zasoby,
3) *Void Aglet.run()* – wywoływanej zawsze wtedy, kiedy tworzona jest instancja agletu bądź jego działanie jest wznawiane.

6. Przykłady zastosowań przenośnych agentów do realizacji transakcji rozproszonych

Analiza cech technologii przenośnych agentów oraz obszaru jej zastosowań, przedstawiona w [8], wykazała, że może ona być z powodzeniem wykorzystywana do rozwiązywania problemów odnoszących się do rozproszonych baz danych. Rozważona została więc możliwość zastosowania przenośnych agentów w tworzonym, przy współudziale autorki, eksperymentalnym systemie zarządzania rozproszoną bazą danych. System ten, opisany w [4, 5], w obecnym kształcie realizuje podstawowe zadania dotyczące rozproszonej bazy danych. Obejmują one:

- przyjęcie i analizę składniową poleceń języka SQL,
- ich dekompozycję zgodnie ze schematem rozproszonej bazy danych zawartym w globalnym słowniku danych; schemat ten dopuszcza istnienie takich typów rozproszeń, jak: fragmentacja pionowa, fragmentacja pozioma oraz replikacja,
- obsługę blokad logicznych [5],
- realizację dwufazowego protokołu wypełnienia transakcji 2PC [2],
- prezentację wyników.

Zadania przewidziane do realizacji przez agentów przenośnych obejmują:

- zarządzanie blokadami logicznymi,
- dostarczenie zestawu operacji do wykonania modułowi odpowiedzialnemu za ich realizację w lokalnej bazie danych,
- uaktualnienie kopii danych w węzłach nie uczestniczących w protokole 2PC.

Analizując cykl życia wymienionych agentów można stwierdzić, że powoływani są oni do życia w różnych fazach pracy aplikacji i w różnych momentach może kończyć się ich działanie: zarządca blokad powstanie przy starcie systemu, drugi z nich, za każdym razem kiedy pojawi się nowa transakcja, ostatni – w reakcji na niemożność udziału węzła w bieżącej transakcji. Okresowo mogą oni przechodzić w stan uśpienia. Jednak od czasu utworzenia powinni reagować na zdarzenia występujące w otaczającym ich środowisku.

Pojawienie się w systemie nowego zdarzenia, którym może być np. rozpoczęcie nowej transakcji lub zatrzymanie pracy systemu, sygnalizowane jest agentom odpowiednim komunikatem. Komunikat ten może zawierać dodatkowe informacje niezbędne do obsłużenia tego zdarzenia, np. operacja należąca do transakcji lub adres węzła, w którym zgłoszono nową transakcję. Zadaniem agenta jest rozpoznanie nadchodzącej wiadomości i wykonanie odpowiedniej dla niej akcji. Przykładowa obsługa takiej wiadomości może wyglądać następująco:

```
public boolean handleMessage(Message msg) {
        if (msg.sameKind("start")) {   // rozpocznij pracę.
          msg.sendReply();
          start();
        } else if (msg.sameKind("stop")) { //zatrzymujemy system
          stop();

        } else if (msg.sameKind("gonext")) { //masz pracę do wykonania  w innym węźle
          try {
                dispatch((URL)msg.getArg());
          } catch (Exception ex) {
                ex.printStackTrace();}
        } else if (msg.sameKind(("clone")){
          try {
                cloneAg(msg);
          } catch (Exception ex) {
                ex.printStackTrace();}
        } else if (msg.sameKind("sleep")) { //odpocznij
          try {
                deactivate(msg);
          } catch (IOException ex) {
                ex.printStackTrace();}
        } else if (msg.sameKind("getInfo")) {
          msg.sendReply( getInfo() );
        } else return false;
        return true;
  }
```

Różnice dla poszczególnych agentów będą polegały na realizacji wymienionych w przykładzie metod: *start(), stop(), cloneAg()* itp. Dla agenta zarządzającego blokadami metoda *start()* będzie między innymi zawierać utworzenie wektora przygotowanego do przechowywania aktualnych blokad oraz wektora z kolejką oczekujących transakcji:

```
Vector locks = new Vector();
Vector queue = new Vector();
```

Agent dostarczający operacje do wykonania będzie potrzebował informację o węzłach zaangażowanych w transakcję:

```
Vector addresses = new Vector();

Message msg = new Message("operacja");
Msg.setArg("SQL", op);
for (int i=0; i<addresses.size(); i++) {
        try {
              AgletProxy proxy = Aglets.createAglet( null, null, "worker",  null);
                        proxy = proxy.dispatch((String)addresses.elementAt(i));
                        String reply = (string) proxy.sendMessage(msg);
                } catch (Exception ex) { ...}
}
```

Podobnie będzie wyglądała sytuacja z agentem odpowiedzialnym za utrzymywanie spójności kopii danych. Agenci tego rodzaju tworzeni będą dla każdego węzła, który nie brał udziału w bieżącej transakcji. Taka sytuacja może zostać wykryta przez koordynatora tej

transakcji, np. kiedy nie uda się operacja utworzenia w tym węźle agenta *workera*. Zadaniem agenta uaktualniającego będzie gromadzenie nadchodzących wiadomości z operacjami do wykonania w nieaktywnym węźle. Ponadto, co jakiś czas, powinien on podejmować próbę przeniesienia się do wskazanego węzła, by tam zrealizować zaległe operacje.

Przykład:

```
URL destination = "http://luna.iinf.polsl.gliwice.pl";
try {
        AgletProxy proxy = Aglets.createAglet( null, null, "updateAg", args);
        proxy = proxy.dispath(destination);
        } catch (exception ex) {...}
public void onCreation (Object args){
        Vector trList = new Vector();
        Dane elem = new Dane();
        if( args != null){
                elem.op = (String) ((Object[]) args)[0];
                elem.nTr = new Integer ((Integer) ((Object[]) args)[1]);
                trList.addIElem(elem);
        }

}
```

Postać metody *cloneAg()* może być następująca:

```
public synchronized void cloneAg (Message msg) {
        try {
             AgletProxy p = (AgletProxy)clone();
             Message m = new Message("go");
             m.setArg("url", msg.getArg("url"));
            m.setArg("destination", msg.getArg(("destination"));
             p.sendMessage(m);
         } catch (Exception ex) {
           ex.printStackTrace();}
  }
```

Zakończenie pracy wymienionych agentów może nastąpić przy zamykaniu systemu bądź w momencie wykonania przez agenta wszystkich powierzonych mu zadań. Zarządca po przyjęciu wiadomości typu "stop" wykona:

```
public void stop() {
    dispose();
}
```

Natomiast agent uaktualniający może sam siebie zwolnić, gdy skończy mu się lista operacji do wykonania:

```
If (trList.elementCount() = 0)
    dispose();
```

7. Wnioski

Wykorzystanie technologii agentów przenośnych do realizacji zadań, w przedstawionym zakresie, nie zwiększa funkcjonalności eksperymentalnego systemu zarządzania rozproszoną bazą danych. Posiada jednak kilka zalet. Agenci mogą być tworzeni w sposób dynamiczny w odpowiedzi na pojawiające się zdarzenia i mogą dostosowywać swoje działania do aktualnie panującej sytuacji. Obejmuje to także zwolnienie samego siebie w przypadku wykonania wszystkich zadań. Pociąga to za sobą możliwość rezygnacji z uruchamiania stacjonarnych programów, które w pętli oczekują na moment, kiedy będą potrzebne do realizacji jakiegoś zadania. Inną zaletą tego rozwiązania jest odciążenie przestrzeni krotek i przeniesienie przechowywanych tam informacji do wewnętrznej reprezentacji agenta. Oczywiście ważnym aspektem pracy agenta jest umiejętność ochrony przed podejrzanymi komunikatami. Problem ten będzie rozwijany w ramach przyszłych badań.

LITERATURA

1. ASDK Homepage: http://www.trl.ibm.co.jp.
2. Bernstein P. et al.: Concurrency Control and Recovery in Database Systems. Addison-Wesley, 1987.
3. Chess D., Harrison C., Kershenbaum A.: Mobile Agents: Are They a Good Idea? IBM Research Report, IBM Research Division, T.J. Watson Research Center, New York, 1995.
4. Chłopek M., Harężlak K., Josiński H.: Implementacja podstawowych mechanizmów zarządzania rozproszoną bazą danych w eksperymentalnym systemie wykorzystującym model wirtualnie współdzielonej pamięci. ZN Pol. Śl. s. Informatyka, z. 32, Gliwice 1997.
5. Chłopek M., Harężlak K., Josiński H.: Sterowanie współbieżnym dostępem do danych podczas realizacji transakcji rozproszonych – rozwój eksperymentalnego systemu rozproszonej bazy danych. ZN Pol. Śl. s. Informatyka, z. 34, Gliwice 1998.
6. Ferber J.: Multi-Agent Systems. An Introduction to Distributed Artificial Intelligence. Addison-Wesley, 1999.
7. Harężlak K.: Realizacja transakcji na kopiach danych w środowisku wirtualnie współdzielonej pamięci. Studia Informatica vol. 22, no. 3 (45), Gliwice 2001.
8. Harężlak K., Josiński H.: Dostęp do rozproszonych danych z wykorzystaniem agentów przenośnych. Studia Informatica vol. 23, no. 4 (41), Gliwice 2002.
9. Sahuguet A.: About agents and databases. CIS-650, 1997. http://www.cis.upenn.edu/~sahuguet/Agents/Agents_DB.pdf.

10. Distributed Computing with Aglets. http:/www.vistabonita.com/papers/DCAglets .
11. Bigus J. P., Bigus J.: Constructing Intelligent Agents Using Java. Wiley, 2001.

Recenzent: Dr hab. inż. Stanisław Wołek Prof. Pol. Rzeszowskiej

Wpłynęło do Redakcji 10 lipca 2003 r.

Abstract

The paper describes a problem of replicated data update with usage of the mobile agents. The article presents the definition of the mobile agent and its features in the Aglets Software Development Kit system (ASDK). The life cycle of the aglet was shown on the Figure 1. The mechanisms of communication of mobile agents, which were created in ASDK system, were presented in the chapter four. The message–passing schema is shown on the Figure 2.

The author discusses the processing of transactions in distributed databases using the mobile agent technology. The discussion is connected with the possibilities of using aglets in the already created experimental distributed database management system. The functions of mobile agents in replicated data management are also considered in the paper. There are following function which are taken into consideration: managing of the logical locks, bringing up to date replicated data that didn't take part in the synchronous update and providing SQL operations to the manager of the replicated data in the local database. The examples of the message handle, clone and dispose procedures are also presented.

Adres

Katarzyna HARĘŻLAK: Politechnika Śląska; Instytut Informatyki, ul. Akademicka 16, 44-101 Gliwice, Polska, k.harezlak@zti.iinf.polsl.gliwice.pl