Agenten in Java: Der technologische Unterbau

Dr. Klaus Dorer, Director Technology Research living systems AG
Christian Dannegger, CTO living systems AG

Abstract

Das Agenten-Paradigma (vorgestellt im vorangegangenen Heft) erlaubt es Systeme zu erstellen, in denen die einzelnen Komponenten autonom und kollaborativ interagieren. Es ermöglicht zudem systemübergreifende Lösungen, bei denen Daten **und** Code mobil sind. Die Wahl der idealen Programmiersprache fällt damit nicht schwer: Java. Dank der Plattformunabhängigkeit und des Serialisierungsmechanismus eignet sich Java in besonderer Weise für die Erstellung von mobilen Agenten. Im Folgenden befassen wir uns mit dem grundlegenden Aufbau von Agentensystemen und deren Umsetzung in Java anhand des *living agents runtime system**.

* *living agents runtime system (LARS)* ist ein Agentenserver der living systems AG und die technologische Grundlage von 30 Internet commerce Plattformen.

(Einleitung)

Nach der eher theoretischen Einführung in das Agenten-Paradigma im vorigen Heft, erläutert dieser Artikel konkret die praktische Umsetzung von Agenten und Agentensystemen in Java. Agentensysteme bestehen prinzipiell aus zwei Teilen: der Laufzeitumgebung für die Agenten und den Agenten selbst (vergleichbar mit Betriebssystem und Programm). Während die Agenten durch die Übernahme von Rollen und deren Verantwortlichkeiten den aktiven Teil eines Agentensystems übernehmen, stellt die Laufzeitumgebung im Wesentlichen die Infrastruktur für das Versenden von Nachrichten, die Mobilität der Agenten, sowie Tools zur Verwaltung der Agenten zur Verfügung.

Agenten

Während für das Design von Agentensystemen neue Verfahren entwickelt werden (siehe Artikel im vorangegangenen Heft), kann der Aufbau eines Agenten mit herkömmlichen, objektorientierten Methoden dokumentiert werden. So zeigt Abbildung 1 den schematischen Aufbau eines reaktiven, also auf Änderungen in der Umgebung reagierenden Agenten, sowie den Aufbau eines proaktiven, in die Zukunft vorausschauenden Agenten.

Ein wesentliches Merkmal von Agenten ist, dass sie situiert, d.h. in ihre Umgebung eingebettet sind. Entsprechend besitzen Agenten auf unterster Abstraktionsebene **Wahrnehmungen**, die den Zustand der Umgebung beschreiben, sowie **Aktionen**, mit denen der Agent die Umgebung ändern kann. Aufbauend auf den Wahrnehmungen und Aktionen besitzen sowohl reaktive als auch proaktive Agenten Verhaltensregeln, die bei proaktiven Agenten zudem die zu erwartenden Effekte einer Aktion repräsentieren müssen.

Proaktive Agenten müssen schließlich auch **Ziele** repräsentieren können, um die Fähigkeit zu erlangen, zukünftige Situationen zu bewerten, sowie Pläne, um die Ziele zu erreichen.

living systems AG - Tel.: +49 (0)771/8987-0 - Fax: +49 (0)771/8987-100 – www.living-systems.com

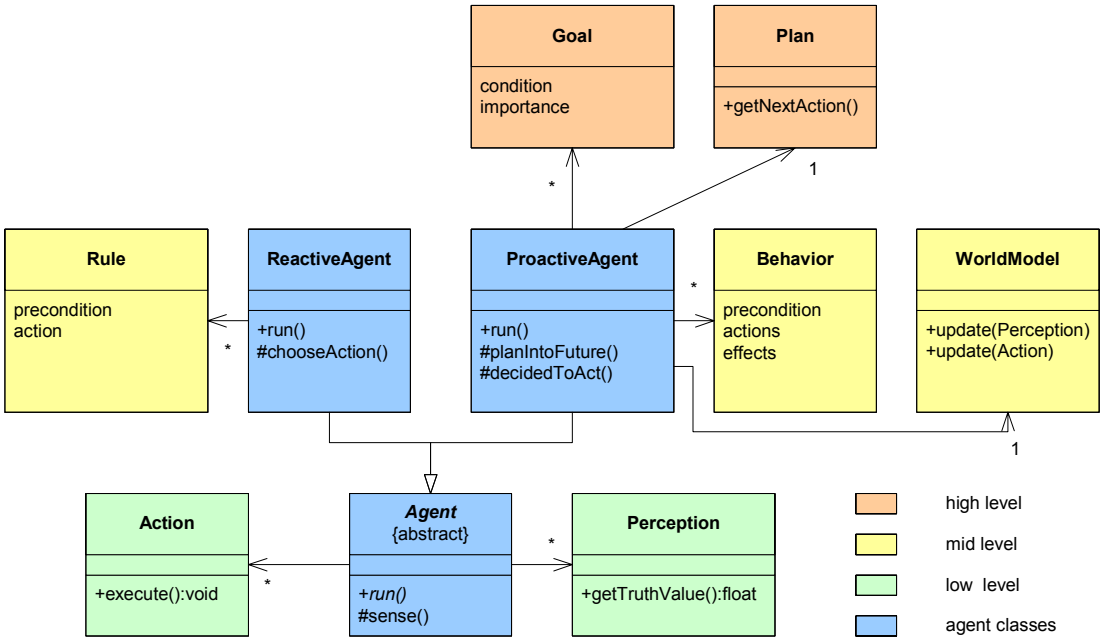

Abbildung 1: Aufbau eines reaktiven und proaktiven Agenten.

Wie bereits im vorigen Artikel beschrieben, gelten Autonomie, Kollaboration und Mobilität als drei wesentliche Eigenschaften von Agenten. Im Folgenden wird beschrieben, wie diese Eigenschaften mit Hilfe von Java umgesetzt werden können.

Autonomie und Zielorientierung

Agenten sind in der Lage, selbständig Entscheidungen zu treffen. Dabei unterscheidet man zwei Arten von Agenten: reaktiv und proaktiv.

Reaktive Agenten nehmen ihre Umgebung wahr und wählen eine angemessene Aktion. Die Aktionsauswahl basiert ausschließlich auf den Wahrnehmungen und einer Menge von Verhaltensregeln (Programmbeispiel 1). Diese einfache Kontrollschleife hat den Vorteil, dass der Agent sehr schnell auf Änderungen in der Umgebung reagieren kann. Beschränken sich die Wahrnehmungen des Agenten auf das Empfangen von Nachrichten, kann der Aufruf von `sense(environment)` sogar blockierend sein, was zusätzlich Rechenkapazität spart.

```
Environment  environment;
Rules        rules;
while (agentIsAlive) {
     perception  = sense(environment);
     action      = chooseAction(perception, rules);
     environment.applyAction(action);
}
```

Programmbeispiel 1: Kontrollschleife eines reaktiven Agenten.

Proaktive Agenten sind demgegenüber in der Lage, vorausschauend zu handeln. Dazu müssen sie zum einen die Effekte ihrer eigenen Aktionen kennen, um zukünftige Situationen antizipieren zu können. Zum anderen müssen proaktive

living systems AG - Tel.: +49 (0)771/8987-0 - Fax: +49 (0)771/8987-100 – www.living-systems.com

Agenten Ziele repräsentieren, um günstige von weniger günstigen zukünftigen Situationen unterscheiden zu können. Typisch ist für proaktive Agenten weiterhin, dass sie die Umgebung intern in einem Weltmodell repräsentieren. Dadurch können Wahrnehmung und Handlung unabhängig voneinander in eigenen Threads ablaufen. Das Weltmodell wird außerdem dazu verwendet, zukünftige Situationen zu repräsentieren.

Wie man sich denken kann liegt die Komplexität der Proaktivität zum einen in der Methode `planIntoFuture()`, die aus der Vielzahl der möglichen Aktionsfolgen diejenige bestimmen muss, die die meisten und wichtigsten Ziele mit möglichst hoher Wahrscheinlichkeit erreicht. Zum anderen hat die Methode `decidedToAct()` die schwierige Aufgabe zu entscheiden, ob es sich lohnt, nach einem besseren Plan zu suchen, oder ob die Situation ein sofortiges Handeln notwendig macht.

```
Environment  environment;
WorldModel   worldModel;
Goals        goals;

// perception loop
while (agentIsAlive) {
     perception = sense(environment);
     worldModel.update(perception);
}

// action loop running in a different thread
while (agentIsAlive) {
     plan = planIntoFuture(worldModel, goals);
     if (decidedToAct()) {
         action = plan.getNextAction();
         environment.applyAction(action);
         worldModel.update(action);
     }
}
```

Programmbeispiel 2: Kontrollschleife eines proaktiven Agenten.

Laufzeitumgebung

Die beiden Eigenschaften Kollaboration und Mobilität von Agenten werden teilweise bzw. ganz von der Laufzeitumgebung abgedeckt. Die Laufzeitumgebung, auch Agentenserver genannt, stellt die grundlegenden Dienste für die Agenten zur Verfügung. Zwei wesentliche Dienste sind dabei das Versenden von Nachrichten sowie das Versenden von Agenten. Die Verwaltung und Ausführung solcher Dienste wird in der Regel von sogenannten Systemagenten gekapselt. Beim Agentenserver *LARS* sind das für das Zustellen von Nachrichten der `AgentMessageRouter` und für das Verwalten und Versenden von Agenten der `AgentManager`. Einen genaueren Überblick wird der vierte Artikel dieser Serie liefern.

Kollaboration

Die Kollaboration von Agenten basiert in der Regel auf der Kommunikation zwischen den Agenten. Die Laufzeitumgebung sorgt dafür, dass Nachrichten auch

living systems AG - Tel.: +49 (0)771/8987-0 - Fax: +49 (0)771/8987-100 – www.living-systems.com

dann den Empfänger-Agenten erreichen, wenn dieser sich beispielsweise gerade auf einer anderen Plattform befindet (siehe Programmbeispiel 3). Während die Weiterleitung von Nachrichten auf der lokalen Plattform mittels einfachem Methodenaufruf erfolgt, werden Nachrichten zu anderen Plattformen mit Hilfe von pure Sockets, secure Sockets, per RMI oder anderen Protokollen transparent für den Agenten zugestellt.

Die Kommunikation der Agenten ist dabei in der Regel asynchron, d.h. nach dem Versenden einer Nachricht kann ein Agent weiterarbeiten ohne auf eine Antwort zu warten. Das setzt voraus, dass ein Agent eine Messagebox besitzt, die die Antwort oder andere eingehende Nachrichten speichert, während der Agent noch beschäftigt ist. Immer wenn der Agent Zeit hat, überprüft er die Messagebox auf neu eingegangene Nachrichten und bearbeitet diese.

Neben der direkten Kommunikation zwischen Agenten stellt *LARS* auch andere Arten der Nachrichtenzustellung zur Verfügung. Agenten können sich beispielsweise als Provider für einen Service anmelden, den andere Agenten ohne direkte Angabe eines Empfängers nutzen können. Sollen mehrere Agenten gleichzeitig angesprochen werden, sind Multicast-, Broadcast- oder Gruppennachrichten möglich.

living systems AG - Tel.: +49 (0)771/8987-0 - Fax: +49 (0)771/8987-100 – www.living-systems.com

```
private synchronized void routeMessage(Message message)
throws RoutingException
{
      Messenger   messenger       = null;
      String      receiver        = message.getReceiver();
      String      amrReceiver;

      if (message.receiverIsInvalid()) {
          throw new RoutingException("bad receiver: " + message);
      }

      if (allMessengers.containsKey(POOL_PREFIX + receiver)) {
          // receiver is an agent pool and the message has to be
          // routed to the least occupied agent of this pool
          receiver = searchLeastLoadedPoolAgent(receiver);
      }

      if (allMessengers.containsKey(receiver)) {
          // agent lives on local platform
          messenger = (Messenger) allMessengers.get(receiver);

      } else if (allForwards.containsKey(receiver)) {
          // there exists a forward to another platform
          amrReceiver = (String) allForwards.get(receiver);
          if (allMessengers.containsKey(amrReceiver)) {
              messenger= (Messenger)allMessengers.get(amrReceiver);
          }

      } else {
          // agent is not local and there is no forward: try to
          // send the message to the receiver's home platform
          String agentHome = getHomePlatform(receiver);
          if ((agentHome != null) && (agentHome.length() != 0) &&
              (! homePlatform.equals(agentHome))) {
              // agent's homePlatform is known and not this one
              amrReceiver = AGENT_MESSAGE_ROUTER + agentHome;
              if (allMessengers.containsKey(amrReceiver)) {
              messenger= (Messenger)allMessengers.get(amrReceiver);
              }
          }
      }
      if (messenger != null) {
          // a messenger was found: pass the message to it
          messenger.processMessage(message);

      } else {
          // no messenger found... tell the sender
          Message replyMessage = new Message(AGENT_NOT_AVAILABLE,
              message.getSender(), Message.LARS_INTERNAL, message);
          routeMessage(replyMessage);
      }
}
```

Programmbeispiel 3: Routing einer Nachricht durch die Laufzeitumgebung

living systems AG - Tel.: +49 (0)771/8987-0 - Fax: +49 (0)771/8987-100 – www.living-systems.com

Mobilität

Ein weiterer Dienst der Laufzeitumgebung besteht im Versenden und Empfangen von Agenten. Besonders bei der Mobilität von Agenten kommen die Vorzüge von Java zum Tragen. Durch die Plattformunabhängigkeit ist ein auf Java basiertes Agentensystem weder an ein Betriebssystem noch an spezielle Hardware gebunden. Der Agentenserver *LARS* ist beispielsweise sowohl auf Unix und Linux als auch auf allen Windows-Betriebssystemen im Einsatz. Agenten können problemlos zwischen diesen Systemen hin und her wandern. Der in Java eingebaute Serialisierungsmechanismus erleichtert weiterhin dieses Migrieren der Agenten von einer Plattform zu einer anderen wesentlich (Programmbeispiel 6). Das Agentensystem muss lediglich das Protokoll für die Migration zur Verfügung stellen (Programmbeispiel 4, 5). Die ‚Verpackung' der Agenten übernimmt Java.

Das dargestellte Protokoll behandelt die grundlegenden Mechanismen der Migration. Dazu gehört zum einen das Starten und Stoppen des Agenten. Zum anderen das Anmelden bei der neuen und das Abmelden von der alten Plattform. Dadurch wird gewährleistet, dass der Agent weiterhin Nachrichten empfangen kann. In offenen Systemen müssen dann noch entsprechende Sicherheits-mechanismen eingeführt werden, um den Empfang von gefährlichen oder schädlichen Agenten zu verhindern bzw. den Handlungsspielraum eintreffender Agenten zu beschränken.

living systems AG - Tel.: +49 (0)771/8987-0 - Fax: +49 (0)771/8987-100 – www.living-systems.com

```
protected void interpretStartMigration(Message message)
{
      String receiver = (String) message.getContent();
      String replyId  = message.getReplyId();
      String agentName= message.getSender();

      MobileAgent travellingAgent;

      // check if the agent exists and is mobile
      if (!allAgents.containsKey(agentName)) {
          /* error */ }
      if (!allAgents.get(agentName) instanceof MobileAgent) {
          /* error */ }

      // send a serialization in process message to the orderer
      sendMessage(SERIALIZING_AGENT, receiver, agentName);

      // disconnect travelling agent from this platform
      travellingAgent = (MobileAgent) allAgents.get(agentName);
      travellingAgent.disconnect();

      try {
          // serialize travelling agent
          String serializedAgent = encodeAgent(travellingAgent);

          // serializedAgent should contain the travelling agent's
          // state of non-transient data as bytecode
          if (serializedAgent != null) {
              // compose the migratingAgent Message
              Map contentHash = new HashMap();
              contentHash.put(AGENT_STRING, agentName);
              contentHash.put(CODE_STRING, serializedAgent);

              //remove agent from list
              allAgents.remove(agentName);
              sendMessage(MIGRATING_AGENT, receiver, contentHash);

              // terminate and remove agent
              travellingAgent.terminateAction();
              travellingAgent = null;
          } else { /* error */ }

      } catch (IOException ioex) { /* error */ }

      // if the agent could not be deleted, re-connect it
      if (!allAgents.containsKey(agentName)) {
          allAgents.put(agentName, travelledAgent);
      }
      travellingAgent.connect();
      travellingAgent.setRunLevel(AgentTemplate.RUN_LEVEL_RUNNING);
      sendMessage(SERIALIZE_FAILED, receiver, agentName);
}
```

Programmbeispiel 4: Verschicken eines Agenten von einer Plattform.

living systems AG - Tel.: +49 (0)771/8987-0 - Fax: +49 (0)771/8987-100 – www.living-systems.com

```
protected void interpretMigratingAgent(Message message)
{
     String  sender      = message.getReceiver();
     String  receiver    = message.getSender();
     String  replyId     = message.getReplyId();
     HashMap content     = message.getHashContent();

     // get the agent's name from the migration message
     String agentName = message.getStringContent(NAME_STRING);
     String agentCode = message.getStringContent(CODE_STRING);

     try {
         AgentTemplate decodedAgent = decodeAgent(agentCode);

         // only a mobile and non-existing agent is accepted
         if (!decodedAgent instanceof MobileAgent) {
             /* error */
         }
         if (allAgents.containsKey(decodedAgent.getAgentName())) {
             /* error */
         }

         MobileAgent travelledAgent = (MobileAgent) decodedAgent;

         // agent built successfully. Now it is re-initialized,
         // added to the active agent list and started
         travelledAgent.reInitializeAction();
         allAgents.put(agentName, travelledAgent);
         travelledAgent.start();

         sendMessage(AGENT_BUILD, receiver, replyId, agentName);
         return;

     } catch (IOException ioex) { /* error */
     } catch (ClassNotFoundException cnfex) { /* error */
     } catch (NullPointerException npex) { /* error */ }

     sendMessage(AGENT_NOT_BUILD, receiver, replyId, content);
}
```

Programmbeispiel 5: Empfangen eines Agenten auf einer Plattform.

living systems AG - Tel.: +49 (0)771/8987-0 - Fax: +49 (0)771/8987-100 – www.living-systems.com

```
private String encodeAgent(Agent agent)
throws IOException
{
      String encoded = null;

      ByteArrayOutputStream   baos = new ByteArrayOutputStream();
      ObjectOutputStream      oos = new ObjectOutputStream(baos);

      // serialize the agent: Java does this for us
      oos.writeObject(agent);
      oos.flush();

      // encoding has to be done, because no special characters
      // are allowed, if plain sockets are used for migration
      encoded = base64encoder.encodeBuffer(baos.toByteArray());

      oos.close();
      baos.close();

      return encoded;
}
```

Programmbeispiel 6: Die 'Verpackung' von Agenten übernimmt der in Java eingebaute Serialisierungsmechanismus.

Fazit

Während in den Anfängen der Agententechnologie die Erstellung eines Agentensystems noch Haken und Ösen hatte, brachte die Einführung von Java insbesondere durch Plattformunabhängigkeit, Multithreading und Objekt-Serialisierung deutliche Erleichterungen mit sich. Dem Entwickler bleibt mehr Zeit, sich um die wichtigen Aspekte eines Agentensystems, dem Design der Rollen und Verantwortlichkeiten oder der Realisierung proaktiven Verhaltens zu kümmern. Auch wenn wir hier aus Platzgründen nur Teilbereiche eines vollständigen Agentensystems darstellen konnten, hoffen wir dennoch, Ihnen die wichtigsten Mechanismen eines Agentensystems näher gebracht zu haben.

living systems AG - Tel.: +49 (0)771/8987-0 - Fax: +49 (0)771/8987-100 – www.living-systems.com