Distributing Computing

Cooperative Computing with Mobile Agents

Boudewijn Rempt <boud@valdyas.org>
David Mertz, Ph.D. <mertz@gnosis.cx>
July 2002

Like a number of technologies, agents have suffered the burden of too much good press. In the business press, one sometimes encounters an ebullient vision of future software agents that can handle your personal affairs for you--everything from shopping for you, to fixing your computer, to planning your schedule. Despite the sci-fi appeal of such clever programs, that is not what agents actually do (at least not now). Instead, agents are really just a strategy and some protocols for distributing computer resources (CPU cycles, disk space, database connections, user I/O, etc)--much like other technologies this column has touched on, albeit with a somewhat different emphasis. For this installment, David Mertz has invited Boudewijn Rempt to share his expertise in agents, gleaned as a developer for Tryllian's Agent Developer Kit.

About This Series

In the ebbing paradigm of stand-alone personal computing, a user's workstation comprises a collection of *resources* that are needed to run an application: disk storage for programs and data; a CPU; volatile memory; a video display monitor; a keyboard and pointing device; perhaps peripheral I/O devices like printers, scanners, sound systems, modems, game inputs, and so on. Even since the 1980s, it has been common for such personal computers to also have network capabilities, but a network card has largely been just another sort of I/O device in the "traditional" paradigm.

"Distributed computing" is a buzz-phrase that has something to do with providing more diverse relationships between computing resources and actual computers. Different resources can enter into different sorts of relationships. The protocols and programs that distribute what were basically hardware resources of PC applications make up only part of the distributed computing picture. At a more abstract level some much more interesting things can be distributed: data; information; program logic; objects. In the end, however, what is ultimately shared between distributed computers are sets of responsibilities. One computer promises another that under certain circumstances it will send some bits that meet certain specifications over a channel. These promises or contracts are rarely firstly about particular configurations of hardware, but are almost always about satisfying functional requirements of the recipients.

This column examines the requirements and responsibilities of computers, in practical scenarios. Each installment will discuss specific technologies as ways of getting groups of computers to do what is mutually asked of them.

Introduction to Agents

For the last decade a buzz has surrounded the advent of ubiquitous mobile agents. These smart software components would travel from computer to computer, autonomously performing useful work for their masters.   This idea has an understandable fascination to it, but actual agent applications have been rather less impressive. The so-called *agents* in a product like Tivoli, for example, fall short of popular imagination--they are neither intelligent nor autonomous, and not very mobile either. The fault here is not with existing software applications, but with misconceptions about what software can really do (at least currently). A better way to understand agent software is as simply another strategy for distributing computing resources, not so much different in kind from technologies this column has discussed previously.

You can develop real agent-based software today, and there are already good reasons to do so. We expect that over time, more applications of mobile agents will arise. There are numerous projects that have created agent development toolkits, and even a few companies. Co-author Boudewijn Rempt works for one of those: Tryllian, and it is the Tryllian toolkit that he has worked on for the past three years that we shall use for the examples in this article. You can download the toolkit from <http://develop.tryllian.com>. Non-commercial use is free.

A piece of software can be called an *agent* if it can perform a task with a measure of independence and intelligence. A mobile agent must be able to travel from one system to another, taking with it all its data. In terms of mobility--but not really intelligence--the *InsecureAgent* example that was presented in this column's second installment on Python Remote Objects (Pyro) is a good example. Pyro is software for distributing *objects* rather than *agents*, but the distinction is drawn in shades of gray. In the Pyro *InsecureAgent* system, the server contained a callable `.runtask()` method that accepted an object as an argument, and called the `.run()` method of the passed object. It is not hard to imagine this arrangement extended such that every node on a network of machines ran such an *InsecureAgent* server--nor would it be at all difficult to program such an arrangement in Pyro.

The problem with a simplistic extension of our earlier *InsecureAgent* example is that it unsafe to allow foreign code to run on your computers. The difference between an agent and a virus is small; and Pyro provides only basic hooks for validation of remote code, not a fine-grained system. Mobile code needs to be executed in a safe environment--an agent runtime environment.

Under a proper design, whenever an agent wants to execute a certain action, the runtime environment checks whether that action is allowed. For example, if an agent is too resource intensive, then the environment will reschedule it with a different priority. Tryllian's ADK uses the Java virtual machine as the basic sandbox, and extends the existing Java security mechanism with custom classloaders, security policies and a restrictive scheduler. *Narval* uses Python's facilities to achieve a similar result.

As a rule, agents are quite autonomous, and all cooperation between agents is achieved using asynchronous messaging. Thereby, agents are very loosely coupled. This offers great advantages when upgrading a component, for instance. If a certain computation intensive task is distributed over agents, it is easy to add more agents if more performance is needed. If a certain host is getting too crowded, agents can move to other hosts on their own initiative, and continue cooperating with their peers on the other host.

Agent applications, especially those that allow self-directed relocation of agents, can be fine-grained and location transparent. The most common term for applications like these is a *swarm*. *Grasshopper*, an agent system that appears no longer to be under active development, was designed for this kind of application. Links to a number of Agent Construction Kits can be found at: <http://www.ece.arizona.edu/~rinda/compareagents.html>

Adding Intelligence

A later installment of this column plans to look at the Narval (Network Assistant Reasoning with a Validating Agent Language) project, but it is worth pointing to the project here. Narval is less dedicated to the security and sandbox issues that Tryllian's ADK address, and more concerned with the reasoning aspects of an agent. If agents are to become significantly more common, both elements will need to be refined and developed.

In our opinion, Narval's documentation somewhat overstates its purpose--a claim is made that it is a system that allows non-technical users to quickly configure personal assistants that will achieve goals on behalf of their users. We agree that Narval allows creation of such assistants/agents, but author David Mertz (who is--in all modestly--rather technically educated, as users go), fully expects to spend a good number of hours in research and development before he can produce any useful agents with Narval.

Caveats aside, programming Narval is quite different from traditional programming styles. Aside from a graphic interface for development--shared by some IDEs in traditional languages, like Visual Basic or Delphi--Narval also requires a different way of thinking. Instead of writing single line commands and calls in a language, one assembles steps into recipes. Steps are themselves fairly high-level collections of programmatic action, but so might be calls to library functions. The intelligence in Narval comes out of the way steps are arranged to form recipes. One step can lead to multiple others, and a step can depend upon others either as a requirement or an option. Exactly what information needs to be determined where, and what step can proceed under what circumstances, is not something a Narval developer explicitly programs. Instead, she simply connects some graphical lines to show the inputs and outputs of various steps (Narval comes with a large standard collection of steps/actions). Dependencies and sequences are resolved and performed by the underlying Narval system.

A nice example of Narval's use is discussed in Nicolas Chauvat's article, *How the French Linux Gazette is Built*. Chauvat had at one time manually coordinated translation and publication of the mentioned journal, then decided to get an agent to do much of the work for him. His Narval application contains a number of rules that relate when articles are available, who has volunteered to perform translation or editing, when deadlines have passed, the past timeliness of contributors, when to actually publish a translation, and a variety of other issues. Basically, all of these tasks are matters of sending and receiving emails, and uploading, moving, and archiving various files. A lot of busy work, but work requiring judgements and knowledge along the way. One *could* program such a system in a traditional programming language, but Narval allows a developer like Chauvat to do less *programming*, and instead simply *specify* the rules to an intelligent agent. And it works.

Applying Agents

A traditional example of an agent application is a personal assistant that crawls a network of agent runtime environments in order to perform a specific task for its owner. Such an assistant agent might retrieve and match information, or convey sensitive information to another machine. Communication between runtime environments can be--at least with Tryllian's ADK--secure: authenticated and encrypted.

Taking advantage of a security model like that in Tryllian's ADK, agents can be constructed that will not deliver their payload unless they are properly accessed. Despite the author's misgivings about the diminution of fair use traditions, such controlled access can--and will--be applied in digital rights management.

Agent Platforms

A platform that executes many--possibly tens of thousands--agents on a host is, to all intents and purposes, a special kind of operating system. Running another operating system--i.e. an agent environment--on top of your regular operating system generally requires substantial hardware resources. These requirements are, we believe, a significant reason why existing, agent applications have not caught on yet, despite their potential utility. A personal assistant that takes over your whole computer is too intrusive for comfort.

Any distributed application is more useful if it can access many platforms; agent environments are no exception to that rule. Most agent environments, including Tryllian's ADK, are written in Java. But much exciting work is being done with Python too: Narval was mentioned, Rover is another. In a Python context, Stackless Python lets you run a huge number of agents, because microthreads are so lightweight that it is possible to give every executing agent one or more (micro)threads--this approach is potentially more versatile than Java's threading (or than standard Python threading too). Although we do not know of any system that has implemented it, the suggestions that co-author David Mertz makes for using Python 2.2+ generators to implement weightless threads might provide another approach in standard Python to support sufficient threading for large agent environments.

Without microthreads, a platform that wants to be able to host a decent numbers of agents--as in tens of thousands--needs to implement its own scheduler. It is clearly impossible with the current state of hardware to run that many full-blown threads, even on a multi-processor machine. Our experience is that five hundred full-thread agents per processor are an approximate limit, regardless which OS platform is chosen on currently available IA32 architectures.

If agent environments are to run on every platform needed, the agents themselves cannot usually be coded in just any language. Usually, the choice of programming languages must be the language the runtime environment is written in. If that language is Java, then it is possible to create agents written in JVM-targeting languages like Jython or NetRexx. In such a case, either each supported platform needs to have the relevant byte-code compiler/library installed, or the agents have to lug the runtime jars with them when traveling to another environment. If Microsoft's .NET environment/virtual-machine becomes sufficiently rich, agent platforms targeted to .NET will probably emerge (thereby supporting whatever platforms .NET supports). Presumably, such a hypothetical agent environment would allow programming agents in any language with a .NET version. For now, however, Java/JVM and Python appear to be the most widely targeted languages/environments for agent development.

Tasks and Messages

How you should go about creating agents and deploying them is very dependent upon your choice of environment. Below, we use Tryllian's ADK. This kit bases agent libraries around the two concepts of tasks and messages. The latter is common; the former is unique to the ADK.

Let us begin with the concept of messages. A standards body, FIPA (Foundation for Intelligent Physical Agents), exists, that regulates the way agents should talk to each other. In theory, agents from different environment should be able to exchange messages using this standard. In reality, this does not work. Agents send and receive messages; when they receive messages, they are allowed to execute code in reaction to that message, for instance to answer it:

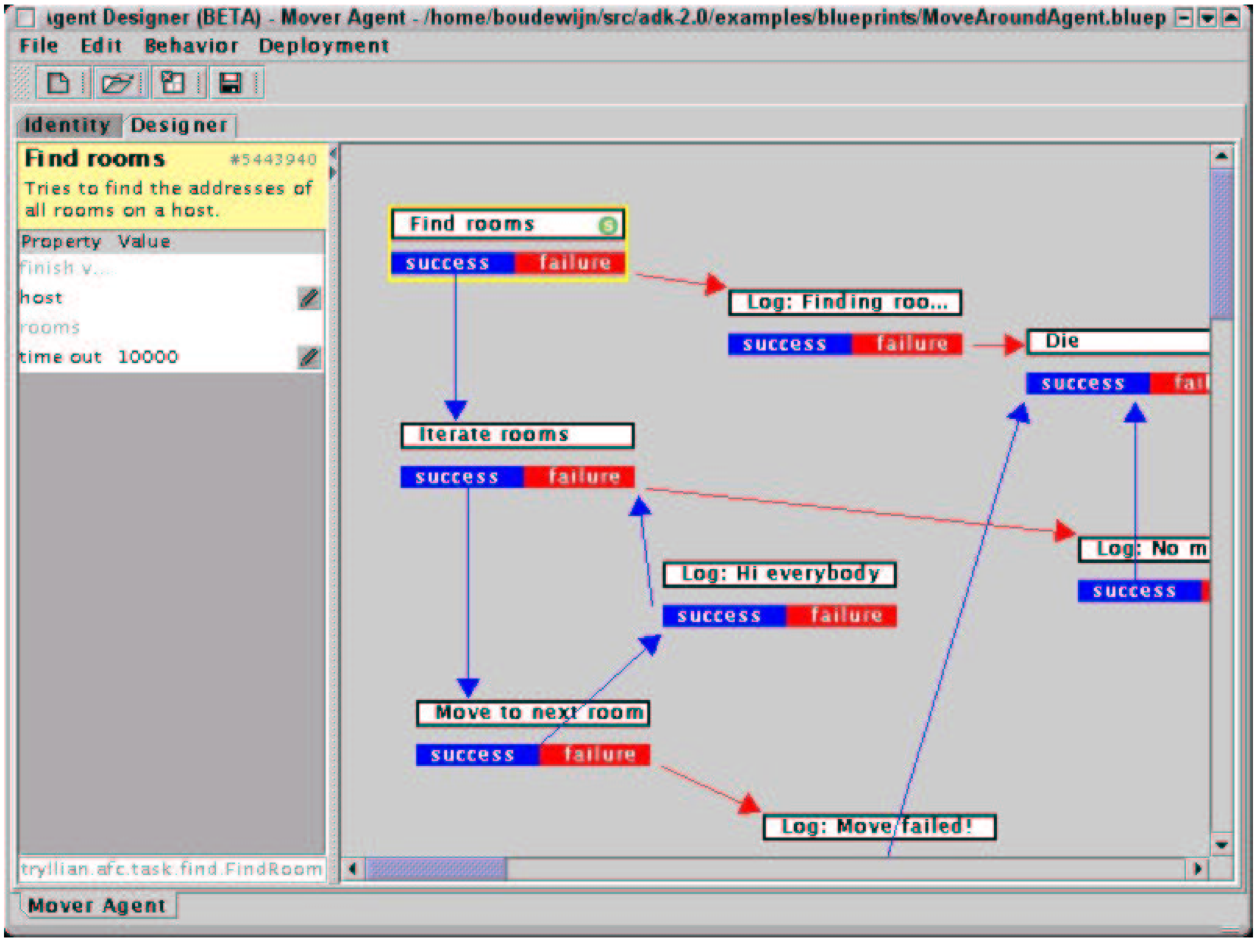
Excerpt from ReceiveGreetingTask.java

```
public class ReceiveGreetingTask
       extends ReactiveTask
       implements MessageHandler {
  private DefaultMessageFilter greetingFilter;
  public ReceiveGreetingTask() {

      // Create filter for messages of type
      // (inform :content(example-message, ...))
      greetingFilter = new DefaultMessageFilter();
      greetingFilter.setPerformative(Performatives.INFORM);
      greetingFilter.setSubject("greeting");
  }
...
public void handleMessage(IncomingMessage message) {
      if (isGreetingMessage(message)) {

        // Get greeting text
        String greeting = extractGreeting(message);
        // Print text
        System.out.println("[Receiving Agent] Message received from "
          + extractSenderName(message));
        System.out.println("[Receiving Agent] It says: '"
                   + greeting + "'");
        System.out.flush();
      }
  }
```

Agents cannot share objects with each other. Even when agents reside in the same runtime environment, all communication between agents is done with messages. Communication between environments is handled by JXTA.

Tasks are an abstraction at an even higher level than class in object-oriented programming. A task is something the agent wants to perform, for instance finding out what environments are available to him. Agents can group and order tasks, creating a high-level flow diagram. The ADK developer can use a GUI tool to do this:

The Visual Agent Designer



Find rooms #5443940
Tries to find the addresses of all rooms on a host.

Property	Value
finish v...	
host	
rooms	
time out	10000

tryllian.afc.task.find.FindRoom

But it is equally easy for a code-oriented developer to create her own tasks and put them in order, for instance in this example of an iterator:

Excerpt from IteratorAgent.java

```
TaskScheduler scheduler = new TaskScheduler();
scheduler.addTask(init, iterator);
scheduler.addTask(iterator, logwriter, done);
scheduler.addTask(logwriter, iterator, null);
scheduler.addTask(done);
addTask(scheduler);
```

An ADK Example

To follow this example, it is best to download the ADK and install it. Agents don't live in a vacuum. With the ADK it's possible to directly access web services like UDDI. In the following example, we create an agent that consults an UDDI directory.

Tryllian has created a number of tasks that help with the use of UDDI. These need to be imported, of course:

```
import tryllian.webservices.registry.UDDIWhitePagesSearchTask;
import tryllian.webservices.registry.UDDIYellowPagesSearchTask;
import tryllian.webservices.registry.UDDIRetrieveWebServicesTask;
import tryllian.webservices.registry.UDDIRetrieveAccessPointTask;
```

The agent will be *transient*; meaning that it won't be stored in a *datastorage* between invocations of the runtime:

```
public class TestUDDITaskAgent
       extends tryllian.afc.agent.Agent
       implements tryllian.are.TransientAgent {
```

Creating an agent is not rocket science:

```
/** Agent class constructor. */
public TestUDDITaskAgent(){
    super();
}
```

The first task is started, when the runtime environment starts the agent:

```
/** Mandatory test method */
public void agentStarted() {
    Task test = new InnerTest();
    System.out.println("\n TestUDDITaskAgent started.");
    addTask(test);
}
```

Using UDDI requires the use of several special tasks, combined is the InnerTest task, which implements the *TaskListener* interface:

```
/**
 * A task that the TestUDDITaskAgent performs.
 */
```

```
public class InnerTest extends DefaultTask implements TaskListener {
    private UDDIWhitePagesSearchTask  whiteuddiTask
            = new UDDIWhitePagesSearchTask();
    private UDDIYellowPagesSearchTask  yellowuddiTask
            = new UDDIYellowPagesSearchTask();
    private UDDIRetrieveWebServicesTask retrieveServicesTask
            = new UDDIRetrieveWebServicesTask();
    private UDDIRetrieveAccessPointTask accesspointTask
            = new UDDIRetrieveAccessPointTask();
```

Every task has a *taskStarted* method. This method is called when the runtime environment schedules the task for execution. We start a *whiteuddiTask* to perform the actual search:

```
/**
 * Sets parameters and adds the UDDIWhitePagesSearchTask.
 */
public void taskStarted() {
    // test something here
    System.out.println();
    System.out.println("\n Starting White Pages Search.");
    whiteuddiTask.setNameSearched(BUSINESS_NAME);
    whiteuddiTask.setUDDIInquiryURL(UDDI_LOCATION);
    whiteuddiTask.addTaskListener(this);
    this.addTask(whiteuddiTask);
}
```

When the subtask is finished, *taskEnded* is called. Here, we handle the results in various complicated ways, and then *succeed*. You can see from this example that you can start new subtasks from *taskEnded*; and if we give the *InnerTest* task as the *taskListener* for those *subTasks*, this *taskEnded* will be called again when a new subtask ends.

```
/**
 * After the UDDIWhitePagesSearchTask is finished,
 * schedules the rest of UDDI tasks.
 */
public void taskEnded(TaskEvent e)  {
    // Handle UDDIWhiteSearchTask results
    if (e.getSource() == whiteuddiTask)  {
         // expect at least one result (because we added it manually -
         // hopefully it stays there for ever)
        List blist =
             whiteuddiTask.getBusinessesInfosKeys(
                            whiteuddiTask.getBusinessInfos());
        boolean hasResult = blist.size() > 0;
        if (hasResult)  {
            System.out.println(
               "[WhitePagesSearch]: »+
               »the keylist of found business entities: ");
            for (int i=0; i<blist.size();i++) {
                 System.err.println(blist.get(i));
            }
            BusinessInfo firstbusiness =
                         whiteuddiTask.getFirstBusinessInfo();
            System.err.println("[WhitePagesSearch]: »+
                               »the first business found: "+
                               firstbusiness.getNameString());
            // check if we found the correct business.
            if (!REGISTERED_BUSINESSNAME.equals(
                            firstbusiness.getNameString())) {
```

```
                System.err.println("WARNING: found business name»+
                                   »is different than expected.");
                System.err.println(
                            "Expected: "+REGISTERED_BUSINESSNAME);
            }
            // store the key for the later test(s)
            REGISTERED_BUSINESS_KEY = firstbusiness.getBusinessKey();

            // Continue with a test of the  Yellow Search task.
            // init the yellowservice task
            this.yellowuddiTask.setUDDIInquiryURL(
       "http://www3.ibm.com/services/uddi/testregistry/inquiryapi");
            // set the category to search for. These are examples of
            // searching for certaisn categores of businesses
            //task.addCategoryNAISC("513000");
            //task.addCategoryUNSPSCv7_3("78.10.15.01.00");

            // search for a company located in NL,
            // using the ISO standards
            // http://www.uddi.org/taxonomies/iso3166-1999-utf8.txt
            this.yellowuddiTask.addCategoryISO3166("NL");
            this.yellowuddiTask.addTaskListener(this);
            this.addTask(this.yellowuddiTask);
            System.err.println("\n Starting Yellow Pages Search.");
        } else  {
            System.err.println(
                "Could not find any businesses matching "+
                BUSINESS_NAME+" at "+UDDI_LOCATION);
        }
    }
    // Handle UDDIYellowSearchTask results
    if (e.getSource() == this.yellowuddiTask)  {
        // expect at least one result, that is why we use a category
        // that should never be empty.
        List blist =
             this.yellowuddiTask.
                 getBusinessesInfosKeys(
                     this.yellowuddiTask.getBusinessInfos());
        boolean hasResult = blist.size() > 0;

        if (hasResult)  {
            BusinessInfo firstbusiness =
                         this.yellowuddiTask.getFirstBusinessInfo();
            System.err.println(
              "[Yellow Pages Search]: the first business found: "
              +firstbusiness.getNameString());

            System.err.println(
              "[Yellow Pages Search]: business keys list found: ");
            System.err.println(blist.toString());

            // This test looks of a service from a known business
            // and service. The business key has been found before
            // using white search task.
            retrieveServicesTask.addTaskListener(this);
            retrieveServicesTask.setBusinessInfoKey(
                REGISTERED_BUSINESS_KEY
                );
            retrieveServicesTask.setUDDIInquiryURL(UDDI_LOCATION);
            retrieveServicesTask.addServiceNameToSearchKeys(
                REGISTERED_SERVICENAME);

            System.err.println(
                "\n Starting the UDDIRetrieveWebServicesTask.");
            System.err.println(
                "[RetrieveWebServices]: Looking for a service named: "
                + REGISTERED_SERVICENAME );
            System.err.println("belonging to a business named " +
                               REGISTERED_BUSINESSNAME );
            System.err.println("In case this test fails, first check"+
              "whether mentioned entries still exist on the UDDI»+
              » server at "+UDDI_LOCATION);
            this.addTask(retrieveServicesTask);
        }
    }
    // Handle the UDDIRetrieveWebServicesTask
    if (e.getSource() == retrieveServicesTask)  {
        ServiceInfos  sinfos =
                      retrieveServicesTask.getServiceInfoList();
        int servicelistsize = 0 ;

        if ( sinfos!=null){
            servicelistsize =
                sinfos.getServiceInfoVector().size();
        }
        if (servicelistsize<=0) {
            System.err.println(
                "[RetrieveWebServices]: could not find»+
                » services for the given business.");
        }
        ServiceInfo serviceinfo =
                    retrieveServicesTask.getFirstServiceInfo();
        if (serviceinfo!=null) {
            System.err.println(
              "[RetrieveWebServices]: found a service with name : \n"
              +serviceinfo.getNameString() );

            System.err.println("[RetrieveWebServices]: UDDI »+
                               »Business key of the business is:\n"+
            serviceinfo.getBusinessKey( ));
            System.err.println("[RetrieveWebServices]: UDDI »+
                               »Service key of the service is:\n"+
                               serviceinfo.getServiceKey());

            // start a Access point Retrieval task.
            this.accesspointTask.addTaskListener(this);
            // add service info key of accesspoint to find
            this.accesspointTask.setServiceInfoKey(
            serviceinfo.getServiceKey()
            );

            this.accesspointTask.setUDDIInquiryURL(UDDI_LOCATION);
            this.addTask(accesspointTask);
            System.err.println(
                "\n Starting UDDIRetrieveAccessPointTask.");
        } else {
            System.err.println(
              "[RetrieveWebServices]: could not find any "+
              " ServiceInfo for the given business");
        }
```

```
            }
            if (e.getSource() == this.accesspointTask )  {
                ServiceDetail sDetail =
                              this.accesspointTask.getServiceDetail();
                if ( sDetail==null){
                    System.err.println(
                            "[RetrieveAccessPoint]: did not get a "+
                            "ServiceDetail for key" +
                            accesspointTask.getServiceInfo().
                            getServiceKey());
                    return;
                }
                System.err.println(
                       "[RetrieveAccessPoint]: Business Services Vector is");
                Iterator i = sDetail.getBusinessServiceVector().iterator();

                while (i.hasNext()){
                    BusinessService item =(BusinessService) i.next();
                    System.err.println(item.getDefaultNameString());
                }
                BusinessService firstService =
                    this.accesspointTask.getFirstBusinessService();
                if ( firstService !=null){
                    System.err.println(
                      "[RetrieveAccessPoint]: first Business Service name - " +
                      firstService.getDefaultNameString());
                    System.err.println(
                      "[RetrieveAccessPoint]: first Business Service "+
                      "description - "+
                      firstService.getDefaultDescriptionString());
                };
                AccessPoint accesspoint =
                    this.accesspointTask.getFirstAccessPoint();
                if (accesspoint!=null) {
                    System.err.println(
                        "[RetrieveAccessPoint]: accespoint URL "+
                        this.accesspointTask.
                        getFirstAccessPointURLString());
                    System.err.println(
                        "[RetrieveAccessPoint]: accespoint type " +
                        accesspoint.getURLType());
                } else {
                    System.err.println(
                        "[RetrieveAccessPoint]: found no accesspoint.");
                }

                // all tests performed, end this task by succeeding.
                succeed();
            }// if
          }
        }
        }
```

Agents in the real world

Creating an application with agents can be much easier than coding everything manually, because agents are very high level components. For instance, Global ID's has used Tryllian's ADK to create a distributed application that ties together databases from many sources. This application first discovers database hosts, then remotely installs an agent environment. Queries over multiple databases are executed by a traveling agent.

Telefuture, a Dutch telephone applications company, has created a dating application. Every hopeful teenager, who uses this application, receives his or her own agent, and those agents try to make a match for their owners. The agents themselves are controlled using SMS messaging. This brings us back to where we started: agents as personal assistants.