SERVICE DISCOVERY IN A MOBILE AGENT API USING SLP

Michel Barbeau
Département de mathématiques et d'informatique
Faculté des sciences
Université de Sherbrooke
Sherbrooke (Québec) CANADA J1K 2R1
E-mail: barbeau@dmi.usherb.ca

Abstract

A mobile agent that performs a network management task needs to access services on visited nodes. In this paper, the integration, for the purpose of service discovery, of the Service Location Protocol in a mobile agent API called Aiglet is presented. The capabilities of Aiglet are illustrated and evaluated using a configuration task requiring discovery of services in wireless local area networks.

Keywords: Mobile agent, network management, service location protocol, software design, wireless local area network.

1 Introduction

The mobile agent (MA) model is a new distributed software development paradigm. It contrasts with the client-server model. Instead of calling operations on servers with some form of synchronization, the user passes its goal to an agent that knows how to handle it without being controlled. MAs are particularly well suited to network management tasks [2].

There are several MA issues that are not well understood and that need further investigation. In particular, there are issues facing the architecture of agent systems [8]. To address such issues, we developed an MA API nicknamed Aiglet that focusses on agent transfer and resource discovery [1]. An important feature of Aiglet is its native multicast agent transfer support. In this paper, we describe the integration of the Service Location Protocol (SLP) [7] to Aiglet resulting in an MA API capable of multicast dispatch and resource discovery. The Aiglet API can be used for the network management tasks requiring resource discovery such as configuration management.

MAs and resource discovery are discussed in Section 2. Our agent API Aiglet and an example wireless configuration problem, solved using Aiglet, are presented in Section 3. Finally, we conclude in Section 4.

2 Resource Discovery

An *agent system* is a context in which agents execute and use resources. It is interesting for an agent system to comprise a mechanism that makes agents capable of discovering resources and services to which they have access. For instance, an MA configuring network interfaces needs to access system services on visited nodes. The code to access the system services may be embedded in the MA code. The agent transports with it all the code it needs. System services are, however, in general platform dependent. The capabilities of such an MA would be limited to a fixed number of platforms for which it has been configured. Support of new platforms cannot be added on-the-fly. In addition, the size of the code of the MA grows with the number of supported platforms. Transporting large code augments the amount of network traffic. Moreover, this approach implicitly assumes that system services are always available, which may not be the case (e.g., the access to a printer). In cases where services are not available, there are no provisions for alterna-

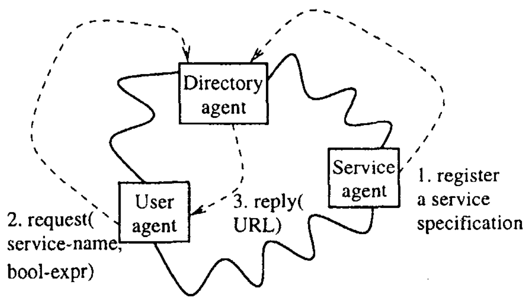
Directory agent
Service agent
User agent
1. register a service specification
2. request(service-name, bool-expr)
3. reply(URL)

Figure 1. Architecture of SLP.

tives. These are serious drawbacks considering the dynamic nature of today's network configurations.

IETF has defined the Service Location Protocol (SLP) for the purpose of discovery of services in a mobile environment [7]. In this paper, we exploit an implementation of SLP available through an API called the Java Naming and Directory Interface (JNDI) [6].

SLP is a naming system associating names with service specifications, in contrast to the DNS that associates names with IP addresses. A service specification consists of an Uniform Resource Locator (URL), i.e., the location of the service, together with an optional list of attributes.

SLP has the concept of service agent (SA), directory agent (DA), and user agent (UA), see Fig. 1. Services of SAs are registered with one or several DAs. To discover a service, an UA sends a request using multicast. The request contains a service name and a boolean expression over values of attributes. For example, SLP can be used to search for a printer with the color printing capability or for the DA itself. If the search is successful, the reply from the DA contains the URL of the service.

Network interface configuration of the hosts in a wireless local area network, e.g. WaveLAN [3] is an example service. According to SLP, an URL giving the location of the configuration service must be defined such as: service:wavelan://dellcom5.u-aizu.ac.jp/wl0
The service is named *wavelan* located at the host *dellcom5.u-aizu.ac.jp* (a host in our research laboratory) for the interface name *wl0*. The JNDI-SLP API actually gives the illusion to the UA that the URL is bound to a Java service object that represents the SA. In reality, the service object is not stored directly. When a service is registered, the URL of a factory object class is attributed to the service specification. When the service is selected by an UA, the factory object class is executed and the service object is produced. Note that the task of this service can also be achieved using a client-server model, i.e., using the Internet's network management protocol SNMP set requests (see Section 3.2).

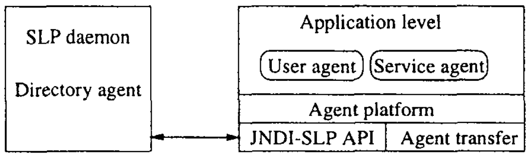

Figure 2. Architecture of Aiglet

3 Mobile Agent API

Aiglet is inspired of IBM's Java Aglets [5] MA system and its architecture is described in Fig. 2. It runs over a Java Virtual Machine (JVM). MAs providing specific services to users are at the *application level*. The *agent platform* is their execution environment. The *agent transfer* module handles migration of agents on the network. MAs act as UAs. The *JNDI-SLP API* handles service registrations and searches. It communicates with a *SLP daemon* which acts as an DA. It stores service specifications. When a service is selected, the API creates a service object, representing the SA and running on the agent platform.

3.1 Application

We illustrate our approach with the case of an MA that is multicast to the nodes of a WaveLAN wireless area network to change the network id (NWID) of network interfaces. A NWID identifies a cell. Change of a NWID may done for configuration purposes. The configuration services are provided by service objects registered with an SLP DA. When an MA arrives to a node, it searches a SLP DA then queries the DA to search for the service objects.

Classes related to the implementation of service ob-

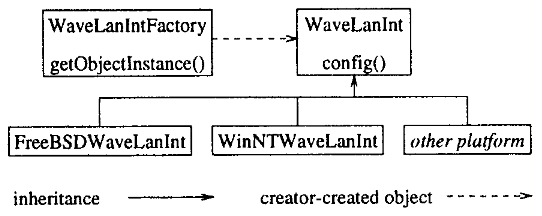

Figure 3. Service object related classes

jects are pictured as boxes in Fig. 3. We assume that network nodes run different operating systems such as FreeBSD, Linux, and Windows. The MA is dispatched to every node, using the multicast capability of our agent API. On every node, it dynamically finds the service required to configure the WaveLAN interface according to the platforms.

There is a factory class, called *WaveLanIntFactory* providing a method called *getObjectInstance()* for creating service objects of the subclasses of class *WaveLanInt*. The subclasses of *WaveLanInt* are defined according to the supported platform. For instance, a service object of class *FreeBSDWaveLanInt* knows how to configure a WaveLan interface on a FreeBSD platform. The service itself is modeled as the method *config()* which is defined as follows in class *FreeBSDWaveLanInt*:

```
1. public String config(String nwid) {
    String data = new String("");
    try {
     String line;
2.   Process s =
     Runtime.getRuntime().
     exec("wlconfig -i "+iName+" -w " + nwid);
3.   BufferedReader in =
      new BufferedReader(
      new InputStreamReader(s.getInputStream()));
4.   while((line = in.readLine()) != null)
      data = data + line  + "\n";
    } catch(IOException e) {
5.   data = e.getMessage(); }
6.  return data; }
```

The service object exploits the command `wlconfig` available on FreeBSD for configuring a WaveLAN interface. Command `wlconfig` is called as a separate process (Line 2). Actual parameters are the local interface name and the NWID. The NWID is a formal parameter of the method, `nwid` (Line 1). The interface name is extracted from the service URL and is passed to the service object when instantiated. It is stored in the data member `iName`. In order to provide feedback to the MA, a handle to the output of the process is obtained (Line 3), read line by line, stored in a variable (Line 4), and returned (Line 6). Whenever an error occurs, the exception message is returned instead (Line 5).

The MA implements two methods: *onCreation()* and *run()*. Method *onCreation()* is as follows:

```
public void onCreation(String nwid) {
 this.nwid = nwid; udispatch(); }
```

It receives and stores, in a private data member, the NWID to apply for the configuration task. Then, it dispatches itself to all nodes of a pre established multicast group. Method *run()* is as follows (with some details abstracted):

```
   public void run() {
    // ref. to service object
    WaveLanInt wLI = null;
    // current host name
    String hostName = null;
1.   <assert the current user is root>
2.   hostName = ...
     try {
3.    SLPServiceProviderContext root =
       SLPServiceProviderContext.
        getSLPServiceProviderContext();
4.     SLPServiceCon-
text serv=root.lookup("wavelan");
5.     NamingEnumeration se = serv.search();
6.     while(se != null && se.hasMoreElements()) {
        SearchResult res = se.next();
        Object obj = res.getObject();
7.      if(obj instanceof  WaveLanInt) {
          wLI = (WaveLanInt)obj;
8.        if(hostName.compareTo(wLI.hostName)==0)
           break;
        }
      }
      if(wLI==null) {
       <report the failure and exit>
      }
    } catch(Exception e) {
     <report the exception an exit>
    }
    try {
9.   String result = wLI.config(nwid);
     <check the result>
    } catch(IOException e) {
     <report the exception an exit>
    }
  }
```

A handle to a service object of class *WaveLanInt* is obtained by querying a SLP directory server. The MA uses the JNDI-SLP API providing search operations. Conceptually, the directory user navigates within contexts and sub-contexts. A *context* is a set of name-to-object bindings. A name can be bound to a sub-context. First, the MA asserts it has the required *root* privileges for this task (Line 1, not detailed). Then, it gets the current host name (Line 2, not detailed). Navigation within contexts starts thereafter. The *top* context for SLP is obtained (Line 3). Then, it looks up for a sub-context bound to the name *wavelan.* This sub-context models the WaveLAN configuration type of service (Line 4). All the bindings within that sub-context are enumerated (Line 5). They are inspected one after the other within a *while* loop (Line 6). The MA selects the one that is an instance of class *WaveLanInt* (Line 7) and which represents the interface of the current host (Line 8). If the agent is running on a FreeBSD platform, it is actually an object of class *FreeBSDWaveLanInt* that is created by the API.

Variable `wLI` stores a reference to this object. Method *config()*, with the NWID, is called on the object (Line 9).

3.2 Evaluation

Fig. 4 shows the latency of the NWID configuration using our approach versus the number of nodes (label *Aiglet*). The time required to do the same task using the SNMP set operations is also plotted (label *SNMP*). It is, however, worth mentioning that the comparison is rather unfair because in contrast to our approach, SNMP does not support resource discovery. With SNMP an user that wants to control a device must have a priori a model of the device parameters, called a management information base or MIB.

The latency using the Aiglet approach goes as high as 0.25 second for three nodes. This can be considered as high for most applications. The reason for consuming so much time is due to the navigation from contexts to sub-contexts performed by JNDI-SLP. In addition to the explicit navigation performed by the MA there is internal navigation performed by the API to give the illusion to the UA that it has access to an object store. A number of multicast messages are sent by the API to complete that navigation. We counted up to 24 multicast messages. Replies are unicast messages, and do not appear on the network if the UA and DA are both on the same node. This heavy use of multicast is a handicap to SLP scalability. Extension of

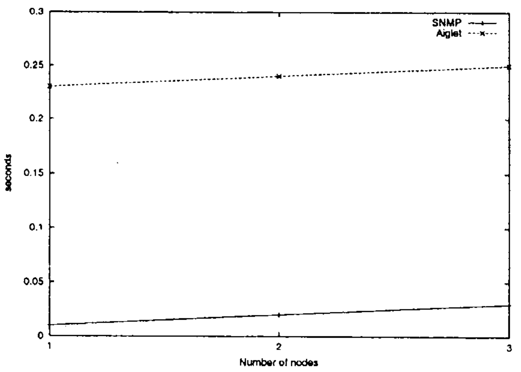

Figure 4. Latency versus number of nodes.

SLP to make it usable on a wide area network by controlling the number of multicast messages is discussed in Ref. [9]. For wireless networks, unicast communication with the DA called Mobile DA, or MDA is suggested [4].

We measured the latency of service registration and we obtained figures close to the ones plotted in Fig. 4.

4 Conclusion

An MA needs resources and services to perform its task. In this paper, we have demonstrated feasibility of resource discovery in an MA system using the Service Location Protocol. Services are registered to a SLP directory when they are available. When they become unavailable, they can be explicitly unregistered. For fault tolerance, a registration has a finite lifetime. Available services can be retrieved and selected by properties with queries. There is also provision for access control, privacy, and authentification.

The performance of the SLP implementation used in this research may be too low for most applications due to the use of multicast communication. We think that the search logic can be reformulated in order to not generate so many multicast messages. The implementation we used behaves as if it forgets the location of the directory from one query to the other. Other implementations of SLP are currently being developed and released. It is expected that they will offer much better performance which will be assessed in our future work.

Acknowledgments

This work was done while the author was a visiting researcher at The University of Aizu, Japan. The author thanks The University of Aizu for supporting this research.

References

[1] M. Barbeau. Implementation of two approaches for the reliable multicast of mobile agents over wireless networks. In *Proceedings of 1999 International Symposium on Parallel Architectures, Algorithms and Networks (I-SPAN'99)*, pages 414–419, Fremantle, Australia, June 1999. IEEE Computer Society.

[2] A. Bieszczad, B. Pagurek, and T. White. Mobile agents for network management. *IEEE Communications Surveys*, 1(1):2–9, Fourth Quarter 1998.

[3] L. M. S. Committee. *802.11 - Local and metropolitan area networks*. Institute of Electrical and Electronics Engineers, Inc., New York, NY, November 1997.

[4] J. Jawanda. Mobile service discovery over wireless networks. Internet Draft, February 1998. 19 pages.

[5] D. Lange, M. Oshima, and O. Mitsure. *Programming and deploying mobile agents with Java Aglets*. Addison-Wesley, 1998.

[6] S. microsystems. The source for java technology - JNDI service providers. http://java.sun.com/products/jndi, 1998.

[7] C. Perkins. Service location protocol. In *ACTS Mobile Networking Summit/MMITS Software Radio Workshop*, Rhodes, Greece, June 1998.

[8] A. Pham and A. Karmouch. Mobile software agents: An overview. *IEEE Communications*, 36(7):26–37, July 1998.

[9] J. Rosenberg, H. Schulzrinne, and B. Suter. Wide area network service location. Internet Draft (work in progress), November 1997.