ELSEVIER

The Journal of Systems and Software 56 (2001) 9–22

The Journal of Systems and Software

www.elsevier.com/locate/jss

SMART mobile agent facility

Johnny Wong [*], Guy Helmer, Venkatraman Naganathan, Sriniwas Polavarapu, Vasant Honavar, Les Miller

Computer Science Department, Iowa State University, C/O Room 205, Atanasoff Hall, Ames, IA 50011, USA

Received 8 August 1999; received in revised form 17 December 1999; accepted 8 February 2000

Abstract

With ever growing use of Internet for electronic commerce and data mining type applications there seems to be a need for new network computing paradigms that can overcome the barriers posed by network congestion and unreliability. Mobile agent programming is a paradigm that enables the programs to move from one host to another, do the processing locally and return results asynchronously. In this paper, we present the design and development of a mobile agent system that will provide a platform for developing mobile applications that are Mobile Agent Facility (MAF) specification compliant. We start by exploring mobile agent technology and establish its merits with respect to the client–server technology. Next, we introduce a concept called dynamic aggregation to improve the performance of mobile agent applications. We, then focus on the design and implementation issues of our system, Scalable, Mobile and Reliable Technology (SMART), which is based on the MAF specification. © 2001 Elsevier Science Inc. All rights reserved.

Keywords: Mobile agent; Mobile agent facility; CORBA; Dynamic aggregation; Client–server

1. Introduction

The growth of Internet has impacted virtually every sector of our society. For example, corporations move to employ the Web internally (Intranet) to create "smart" enterprise, government uses it to reduce costs and provide high quality service and the public at large now has access to a huge and rapidly growing source of information and services of all kinds. From a society point of view, the Web further intensifies the pressure of information overload and from the network point of view, growing Web usage will upset network models and put tremendous pressure on bandwidth requirement and network management (University of Ottawa).

Today's networks pose a barrier to the development of communicating applications. This barrier results from the need for such applications to physically distribute themselves. That is, they should run not only on the computers dedicated to clients, but also on the computers that clients share, the servers. For example, a communicating application that is to provide a forum for buying and selling products necessarily has two parts. A user-interface component in the user's personal communicator gathers information from an individual buyer or seller. A database component in a server records the information and uses it to bring buyers and sellers together (White, 1996).

Most applications involving communications over a network use traditional client–server paradigm in which a connection is established between the client and the server or datagrams are sent across the network. This traditional approach becomes expensive and unreliable when lot of messages have to be sent between the client and the server, i.e., when the application consumes a lot of network bandwidth. In such situations, it is more efficient and reliable to be able to send the client to the server's machine and perform the job locally rather than shouting the commands across the network. This forms the basis for mobile agents. Mobile agents are software agents that have the basic capability to move themselves from host to host and continue execution from the point they stopped on the previous host (Bradshaw, 1997). Additionally, mobile agents can also have the capability to perform functions on behalf of the client, think intelligently (using some AI algorithms), learn and remain persistent.

Mobile software agent that enables the programs to move to the data is a powerful new paradigm that will

[*] Corresponding author. Tel.: +1-515-294-2586; fax: +1-515-294-0258.
E-mail address: wong@cs.iastate.edu (J. Wong).

undoubtedly prove to be a viable option for effective multimedia communications, information gathering and retrieval on the Internet. The main reason for this is the asynchronous, adaptive, and the most important of all, bandwidth-saving nature of mobile agents. The following section discusses this new emerging model of computing and communication paradigm known as the mobile agent infrastructure. The concept of computing using mobile agents is also known as remote programming.

1.1. Mobile agent technology

One definition of term 'agent' means those relatively simple, client-based software applications that can assist users in performing regular tasks such as sorting e-mail or downloading Web pages from the Web, etc. This class of agents is often referred to as 'personal assistant' agents. At the other end of the scale is the concept of sophisticated software entities possessing artificial intelligence that autonomously travel through a network environment and make complex decisions on the user's behalf.

Our definition therefore is the following: a mobile agent is a program that acts on behalf of a user or another program and is able to migrate from host to host on a network under its own control. The agent chooses when and to where it will migrate and may interrupt its own execution and continue elsewhere on the network. The agent returns results and messages in an asynchronous fashion (University of Ottawa).

Mobile agents do not require network connectivity with remote services in order to interact with them and network connections are used for one-shot transmission of data (the agent and possibly its state and data). Results in the form of data do not necessarily return to the user using the same communication trajectory, if indeed the results are to be returned at the originating site. Alternatively, the agent may send itself to another intermediate node and take its partial results with it. Results are delivered back to the user whose address the agent knows.

Today the most common way of implementing distributed applications is through the client–server paradigm. In this model, an operation is split into two parts across a network, with the client making requests from a user machine to a server which services the requests on a large, centralized system. A protocol is agreed upon and both the client and server are programmed to implement it. A network connection is established between them and the protocol is carried out. However the client–server paradigm breaks down under situations dealing with highly distributed problems, slow and/or poor quality network connections, and especially in the maintenance of constantly changing applications.

In a system with a single central server and numerous clients, there is a problem of scalability. When multiple servers become involved, the scaling problems multiply rapidly, as each client must manage and maintain connections with multiple servers. The use of two-tier systems or proxies only moves this problem to the network. It does not eliminate the basic problem. With client–server technology there comes a need for good quality network connections. First, the client needs to connect reliably to its server because only by setting up and maintaining the connection may it be authenticated and be secure. Second, the client needs to be assured of a correct response, since a server can crash anytime between processing the request and sending back the reply. Third, it needs good bandwidth since, due to its very nature, client/server must copy data across the network.

Finally, the protocol which a client and a server agree upon is by its very nature specialized and static. Often, specific procedures on the server are coded in the protocol and become a part of the interface. Certain classes of data types are bound to these procedures and the result is a special network version of an application program interface. This interface is extensible, but only at the high cost of re-coding the application, providing for protocol version compatibility, software upgrade, etc. As the applications grow and the needs increase, client/server programming rapidly becomes an obstacle to change (White, 1996).

Mobile agents overcome all these inherent limitations in client–server paradigm. First and foremost, the mobile agent paradigm shatters the very notion of client and server. With mobile agents, the flow of control actually moves across the network, instead of using the request/response architecture of client–server paradigm. In effect, every node is a server in the agent network and the agent (program) moves to the location where it may find the services it needs to run at each point in its execution. For example, the same agent interacts with the user via a GUI to obtain request keys, then travels to a database server to make its request.

The scaling of servers and connections then becomes a straightforward capacity issue, without the complicated exponential scaling required between multiple servers. The relationship between users and servers is coded into each agent instead of being pieced out across clients and servers. The agent itself creates the system, rather than the network or the system administrators. Server administration becomes a matter simply of managing systems and monitoring local load.

The problem of robust networks is greatly diminished for several reasons. The hold time for connections is reduced to only the time required to move the agent in or out of the machine. Because the agent carries its own credentials, the connection is simply a conduit, not tied to user authentication or spoofing. No requests flow across the connection, the agent itself moves only once, in effect carrying a greater "payload" for each traversal. This allows for efficiency and optimization at several levels.

Last and the most important, no application-level protocol is created by the use of agents. Therefore, compatibility is provided for any agent-based application. Complete upward compatibility becomes the norm rather than a problem to be tackled, and upgrading or reconfiguring an application may be done without regard to client deployment. Servers can be upgraded, services moved, load balancing interposed, security policy enforced, without interruptions or revisions to the network and clients.

In general we can say that the mobile agent model offers the following advantages over traditional client–server models:

- Uses less bandwidth by filtering out irrelevant data (based on the user profile and preferences) at the remote site before the data is sent back.
- Ongoing processing does not require ongoing connectivity.
- Saves computing cycles at the user's computer.
- It is more efficient since the processing moves closer to the data.
- More reliable since the processing does not depend on the continuous network connection.
- Frees the user to log out or migrate since the agent's life is independent of the user's session.

Mobile agent technology is gaining widespread use in many data mining and Internet search applications. Today's Mobile agent technology provides us with tools and languages for fast and easy development of mobile agent applications. There are already several commercial mobile agent platforms available in the market today. The next section discusses some of the important commercially available mobile agent platforms and the advantages of the mobile agent programming facilities these platforms provide.

In this report we will also study a new feature in mobile agents known as dynamic aggregation. In object-oriented systems "aggregation" is defined as a-part-of relationship in which objects representing components of other objects are associated as an assembly (Prentice-Hall). Aggregation can be classified into two parts: static and dynamic. Static aggregation of objects can be achieved through, for example, inheritance (Sun Inc., 1998). Static aggregation in object-oriented programs is formed at the compile time and any irregularities in inheritance or object-containment is detected during the compiling phase. On the other hand dynamic aggregation refers to enhancing the properties of an object at runtime in unforeseen ways. During an object's execution phase, it can form relationship with other objects of unrelated classes to enhance its functionality.

In mobile agents the importance of dynamic aggregation was realized when people started using mobile agents in AI and data mining applications. An agent would typically be required to carry several algorithms and hundreds of rules with it from one host to another, many of those algorithms not being used at all. This would cause the agent code to bloat up and hence would result in more network bandwidth consumption. It would also take more CPU power to pack the agent for transporting it across the network and to unpack the agent after it has been received at the receiving agent system.

The traditional mechanisms of inheritance and polymorphism do not help to solve these problems. Dynamic aggregation allows us to attach new code and data to an agent at the runtime. Looking from a mobile agent's perspective, dynamic aggregation helps in reducing the amount of code that goes along with the agent by allowing the agent to attach extra code on the need basis hence reducing the network bandwidth requirements and also speeding up the process of packing the agent to transfer it from one host to another.

1.2. Project goal

With the emergence of variety of commercial mobile agent platforms, it was recently realized that very soon the mobile agent applications developed on these platforms will start facing interoperability problems. It will no longer be possible to write mobile code that can talk to another mobile code developed on some other mobile agent platform. Because of this, ObjectSpace in conjunction with OMG, came out with a common mobile agent facility (know as MAF) specification in early 1997 (OMG, 1997). Following problems were addressed in the mobile agent facility specification: (White, 1998)

- Interoperability.
- Mobile agents require an environment, in which they can be created, named, authenticated, dispatched to and received from another environment.
- Various languages like Java (Sun Inc.) AgentTCL, Smalltalk need to be supported.
- Various commercial agent implementations need to be supported. For example Aglets (Lange, 1996, IBM Inc., 1997a,b), Concordia (Mitsubishi Electric), Odyssey, Voyager (ObjectSpace Inc., 1997) etc.
- Standard services need to be supported, e.g., naming, authentication, communication, notification, transfer, etc.

The goal of this project is to develop an MAF compliant mobile agent platform (Tham, 1996) and to incorporate features like dynamic aggregation that are still not part of MAF specification. This platform, known as Scalable Mobile and Reliable Technology (SMART), will enable us to write mobile agent applications and run them in environments that are MAF compliant. The applications developed using SMART will be interoperable with other MAF compliant mobile agent technology. We will also incorporate dynamic aggregation facility in it. Although dynamic aggregation is still not a part of the MAF specification, we have realized an early need of this feature to make our mobile agent applications more efficient.

1.3. Roadmap

In Section 2 we will discuss dynamic aggregation and its application in subject-oriented programming. In Section 3 we will present the system design and architecture. A comparison between the SMART and a few commercial mobile agent platforms will also be given. Section 4 will deal with the implementation and testing of SMART system. Section 5 concludes this report and explores some possible areas of future work in our system.

2. Background and related work

2.1. Dynamic aggregation

In this section, we will review the dynamic aggregation facility in detail. Voyager is the only commercial mobile agent platform currently supporting dynamic aggregation feature. The main idea behind incorporating dynamic aggregation in Voyager was to use already existing Java classes that were written before the mobile agent platforms were built. To make objects of such classes mobile, an agent would incorporate those objects as its attachments (known as facets) and move from one site to another hence moving those objects with itself. The objects (carried as the agent's facets) will retain their internal state upon moving from one host to another.

2.1.1. Features of dynamic aggregation

In Voyager, the object to which the other objects are attached dynamically is called a "primary" object and the attached objects are known as "facets". Following features are supported by Voyager's dynamic aggregation:

- A facet class need not be changed or recompiled in order to attach it to a primary object.
- A primary class is not required to be modified or recompiled in order to have facets.
- Any number of facets can be attached to a primary object.
- It is not possible to detach a facet once it is attached to the primary object.
- Voyager does not support attaching a remote facet to a primary object and vice-versa. The code of an attachment must be available on the local host.
- In Voyager one can get hold of a remote reference of a facet in the same way as one can get a remote reference to a primary object.
- Nested facets are not supported in Voyager.
- Cyclic facets are also not supported in Voyager. Probably it is considered dangerous to allow cyclic facets.
- The regular rule in Voyager says that a facet cannot be garbage collected until all the references to the facet as well as its primary object are gone. Voyager provides an istransient( ) functions to override this rule. If this method returns true, the facet object is reclaimed immediately when there are no more references to it. This feature is useful when the facet is stateless and does not need to be associated with the primary object once its work is done. The Voyager Mobility facet is an example of a transient facet.

The details on Voyager's dynamic aggregation API can be found in the Voyager 2.0 user's manual (ObjectSpace Inc., 1997).

2.2. Subject oriented programming

Subject oriented programming is a new concept closely related to dynamic aggregation. Dynamic aggregation can be said to be a practical implementation of subject oriented programming. Subject oriented programming is a program-composition technology that supports building object-oriented systems as compositions of subjects. A subject is a collection of classes or class fragments whose hierarchy models its domain in its own subjective way. A subject may be a complete application in itself, or it may be an incomplete fragment that must be composed with other subjects to produce a complete application. Subject composition combines class hierarchies to produce new subjects that incorporate functionality from existing subjects. Subject-oriented programming thus supports building object-oriented systems as compositions of subjects, extending systems by composing them with new subjects, and integrating systems by composing them with one another (perhaps with "glue" or "adapter" subjects) (IBM Research, 1998).

2.2.1. Practical applications

The flexibility of subject composition introduces novel opportunities for developing and modularizing object-oriented programs. Subject-oriented programming-in-the-large involves determining how to subdivide a system into subjects, and writing the composition rules needed to compose them correctly. It complements object-oriented programming, solving a number of problems that arise when object-oriented technology is used to develop large systems or suites of inter-operating or integrated applications. It is useful in a variety of software development and evolution scenarios, including (IBM Research, 1998):

- Application extension, including unplanned extension.
- Development of application suites by teams operating with varying degrees of independence.
- Development of systems with multiple features, where code for each feature is to be kept separate.
- Development of systems in which it is advantageous for code modularity to mirror requirements structure.
- Unplanned integration of separately written subject-oriented applications.

In C++ programming environment, each subject is written as a collection of C++ class definitions in a namespace, each declaring and defining just what is appropriate for that particular subject. Any standard C++ code can be treated as part of a subject. We see a close relationship between subject-oriented programming and dynamic aggregation facility by realizing the fact that an agent may want to carry only the code that is appropriate for the actions it is going to carry out. Any other irrelevant code (which may be relevant in other scenarios) might have to be left out or attached to the agent on need basis. Hence we see that dynamic aggregation facility helps us to build an agent that is based on its subjective needs.

3. System design and architecture

In this section we will present various design issues involved in implementing SMART and dynamic aggregation feature in it. At the end of this section we will give a comparison between SMART and the three mobile agent platforms discussed in the previous section.

3.1. Design goals

Following are the design goals for SMART:

- *Mobility*: Any programmer wanting to write a mobile code using SMART must be able to do so easily. The application programmer need not bother about understanding and implementing the mobility.
- *Standard compliance*: The system developed should be compliant with the MAF specification.
- *Platform independence*: Since an agent can travel from a machine of one type to machine of another type, it is necessary that SMART be platform independent. This will relieve the programmer of the problems arising due to heterogeneous environments.
- *Performance*: The migration of an agent can be an expensive process especially if the agent is carrying huge amount of data and code. Since nothing much can be done to speed up Java serialization and deserialization we are incorporating dynamic aggregation to improve the performance of SMART.
- *Scalability*: The number of agents roaming around in the system should not be a constraint on the system. The distributed nature of the underlying ORB (in this case Visibroker (Visigenic Inc., 1997)) and Java RMI helps in solving the scalability issues.

3.2. Smart system architecture

SMART is a four tiered architecture that is built on top of Java virtual machine. The lowest layer is the Region Administrator, which is built on Java virtual machine. It manages a set of agent systems and enforces security policies on them. The Finder module at this level offers the region administrators and the layers above it the naming service (see Fig. 1).

The next layer is the Agent System layer. This layer acts as the world of agents allowing them to create, migrate and destroy themselves in this world. This layer can have multiple contexts called places where agents execute. The layer on top of this layer is the agent context layer, also called the Place. This layer provides an execution environment for the execution of agents. Mobile agents, when they migrate, travel between these places. Using agent contexts like places, accessibility policies to resources can be enforced. The top most layer is the Agent Proxy layer. This layer constitutes the mobile agent API which can be used by the applications written in SMART.

The main objects in the MAF system are Agent Proxy, Place, Agent System, Region Administrator and the MAF Finder. All these modules have been implemented in Java. The Agent Proxy has been designed as a Java thread. The application mobile agent must inherit the MobileAgent class to become mobile. The agent proxy communicates with the place server on which the agent resides currently. The place server is designed as an RMI server (non-CORBA object) because of the nature of the information exchanged between the Agent Proxy and the place server. The place server interacts

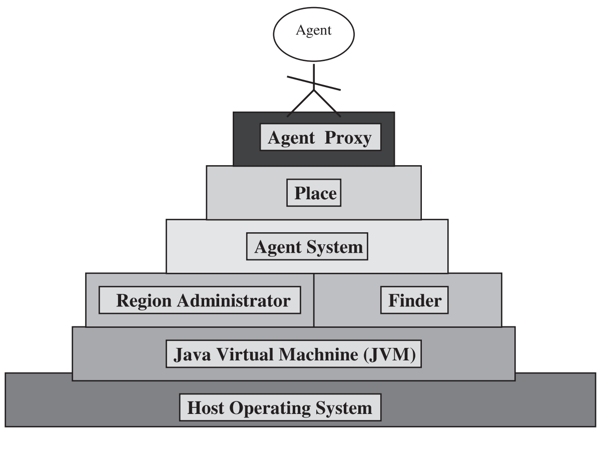

Fig. 1. Four tiered layered architecture of SMART.

with the agent system, to which the agent wants to migrate. The place server may also communicate with its parent agent system for certain agent management operations such as register, locate and unregister. The agent system manages the place server, which runs under it and also communicates with the region administrator, which manages the region to which the agent system belongs. The region administrator enforces security policies on agent systems running under a particular region. The MAF Finder provides a naming service for the above-mentioned components of the MAF system. The Region Administrator is designed as a CORBA (OMG, 1998) object.

3.3. Dynamic aggregation

The idea of using mobile applications to save network bandwidth can be further strengthened by saving the amount of network bandwidth and CPU cycles it takes to pack, transmit the agent's serialized code across the network and unpack and resurrect the agent on the other end of the network. This can be achieved by directly addressing the issues related to methods of agent serialization. Some of the issues include compressing the agent code before transmission, caching the agent code, providing the agent structure information (class information in Java) in advance, etc. We have looked into a new concept called dynamic aggregation in which the agent carries minimum amount of code and data with itself by stripping itself off of unnecessary code and attaching the required code on need basis. Dynamic aggregation will allow a mobile agent to attach code at runtime and hence obviating the need to use a mobile agent that has been programmed and compiled with code ready for all kinds of possibilities. For example, a car mechanic agent need not carry manuals to repair all kinds of cars available on this planet. Instead, the mechanic can pick up only the required manuals or tools depending on the make of the car he is going to repair and the problem at hand.

Dynamic aggregation is useful not only in the mobile agent systems but also in the construction of object-oriented systems in the following ways (ObjectSpace Inc., 1997):

- An object's behavior may be needed to be extended at the runtime in unforeseen ways.
- Behavior must be added to a third-party component whose source is not available.
- Customize an object on the user's need basis.

3.3.1. Requirements

Mobile Agent Facility (MAF) specification does not address the issue of optimizing agent transfer and hence does not discuss the issues concerning dynamic aggregation. As such, a mobile agent platform that would support dynamic inheritance or dynamic aggregation should have the following properties:

- An attachment class should not be required to be modified or recompiled in any way to be attached to a primary class.
- Similarly, a primary class should not be required to be modified or recompiled in any way in order to have any classes attached to it.
- One should be able to attach as many attachments as possible to the primary object.
- It should be possible to detach the attachments whenever needed. Otherwise, the code of a mobile agent would bloat up if it keeps on attaching new objects to it and there is no way to detach them.
- The attachments can be instantiated on any machine regardless of where the primary object is instantiated.
- One should be able to get a remote reference to an attachment object in the same way as one can get a remote reference to the primary object.
- An attachment should be able to have attachments.
- Cyclic attachments should be possible. That is, the primary object of an attachment can be latter's attachment. This may be a controversial feature, which may lead to problems. For example, when somebody tries to find the attachments of a primary object recursively.
- An attachment should not be garbage collected if there are any live (possibly remote) references to either the primary object or the attachment object itself. The vice-versa should also be true.

SMART uses Java serialization to pack an agent for transportation and any object that does not implement the serializable interface cannot be serialized at runtime. Hence the instances of a class that does not implement serializable interface cannot be attached to the agent. Also, not every class instance can become a primary object and start attaching other objects to it. Only those class instances can become primary objects whose class extends MobileAgent class. Without extending MobileAgent class, a class instance cannot get the properties to move. This is a requirement specified in MAF specification and hence cannot be changed. On the other hand, Voyager, which is not MAF compliant, allows any class instances to take the role of a primary object because Voyager supports mobility in the form of dynamic aggregation. Any object can become mobile even if its class does not extend any system class. This can be done by attaching a facet called "mobility" to the object and using the mobility API of the "mobility" facet.

In SMART, dynamic aggregation can be best supported by MobileAgent class because we are giving this feature only to the mobile agents and not to instances of any arbitrary class. The agent will have to maintain a list of all its facets. Facets are those objects that are attached to the mobile agent at runtime. The agent will have to maintain the properties of all the facet objects. It is best to implement all the dynamic aggregation APIs in the MobileAgent class as it will make the API readily available to the mobile agents. But it may not be possible to incorporate all the code in MobileAgent class itself. For example, while transferring the agent from one host to another, some of its facets may not be transferable (due to serializability constraints) and may have to be left out at the source site. In that case, they will either have to be taken care of by the source place server or the source agent system so that the agent can maintain a remote reference to the left out facets.

Incorporating dynamic aggregation directly affects the mobility of an agent in SMART in the following ways:

- Some of the agent's facet objects may not be serializable. Such objects cannot be serialized and hence cannot be transported with the agent to the next site.
- Some of the agent's facet objects may be server objects that will become meaningless if transported to another machine. Even if they are transferable to other machines they may not be able to use the same port number they were previously using. Syntactically it is not feasible to determine if an object is being run as a server.
- A facet object may have open network connections at the time the agent (primary object) is packed and sent to the next site. In such cases, the connections will have to be terminated or some forwarding agent should take care of forwarding the messages.

A simpler solution to these problems would be to filter out such facets from being sent along with the agent. The agent will then have to maintain a remote connection with its facets that were left out at the previous site. In that case, any use of such facets by the agent will have to use the network bandwidth and hence defeats the whole purpose of using dynamic aggregation.

Availability of class information for deserialization of agents and facets is also an important issue. We may assume that the required class information is available before hand at all the sites the agent is going to visit. This assumption may work well when there is only one class to be deserialized but in the case of dynamic aggregation any number of objects (which may all be instances of different classes) can be attached as facets to the mobile agents. All these objects will have to be serialized and deserialized and resurrected as part of the agent transfer from one site to another. In that case it will be impractical to assume that the class information of all the facet objects is available before hand at all the sites. To handle this situation, SMART must have some kind of network class loading facility incorporated into it that will help in downloading the class information about the agent and the facet objects. In the SMART design, the place server is responsible for serializing the agent and it should have access to the class information of the agent and the facet objects. Hence it will be appropriate to make the source place server responsible for providing the class information to the destination network class loader (Javaworld). The place server will be the most appropriate place for housing the network class loader.

3.4. Comparison

This section compares the most successful commercially available agent systems with SMART. The comparison is based on the most desirable features (ObjectSpace Inc., 1997).

Table 1 shows that Voyager is comparatively more advanced than the other three systems. The dynamic aggregation in SMART allows detaching the facets whereas it is not possible in Voyager. In Voyager any object can be made mobile whereas SMART has the restriction of extending from MobileAgent class to make an object mobile. None of the above systems, except for SMART, has their source code publicly available.

4. Implementation and testing

In this section we will discuss the implementation details of SMART with respect to agent model and mobility, dynamic aggregation, network class loading and tracking the agent. Next, we will present the organization of the system code and a brief description of classes and the API. Finally, the testing methods will be discussed.

4.1. Implementation details

4.1.1. Agent model

A mobile agent in SMART has been modeled as a Java thread. This enables the place servers and the agent systems to have a better control over the agent. The agent can be started, suspended, resumed and stopped easily. The agent cannot run any other threads within itself. Every time a mobile agent reaches a target, it is started as a new thread in the target. When it is about to depart it is suspended in the source system, serialized and sent to the target system. Threading agents like this might cause a lot of consistency problems since the tables maintained by the various modules are going to be updated by various threads. SMART has to handle all these situations effectively and efficiently. For example, more than one agent might try to enter/leave the same target/source or more than one agent might execute in a place server concurrently. Thus SMART has to accept, execute and transfer agents concurrently.

4.1.2. Implementation of mobility

Mobility in SMART is implemented primarily by Java's object serialization techniques. Using Java's object serialization it is possible to capture the data and

Table 1
Comparison of SMART to commercially available agent systems

Feature	Voyager	Odyssey	Aglets	SMART
Remote messaging				
Creating agents remotely	Yes	No	No	Yes
Sending Java messages remotely	Yes	No	No	Yes
Sending Java messages to mobile agents	Yes. Transparent to application agents	No	No	Yes but explicit
Messaging modes between agents	Yes	No	Yes	No
Life spans	Five modes	Explicit	Explicit	User controlled
Directory service				
Mobile directory service	Yes. Integrated.	No	No	No
Federated directory service	Yes	No	No	No
Object mobility	Yes	No	No	Yes
Agent mobility				
Moving to a program	Yes	Yes	Yes	Yes
Moving to an object	Yes	No	No	Yes
Dynamic aggregation	Yes	No	No	Yes
Itineraries	No special API needed	Special API needed	Special API needed	No special API needed
Persistence				
Object persistence	Yes	No	No	No
Agent persistence	Yes	No	No	No
Database integration	Integrated with ORACLE, SYBASE, etc.	No	No	No
Database independence	Yes	N/A	N/A	N/A
Flushing/auto-loading	Yes	No	No	No
Scalability	Yes	No	No	Yes
Multicast messaging	Yes	No	No	No
Distributed events	Yes	No	No	No
Applet connectivity	Full	Restricted	Restricted	No
Security	Security Manager	No special mechanism	Security Manager	No special mechanism

state of the agent as a sequence of bytes. This sequence of bytes can be transmitted over the network and can be de-serialized to get back the original agent. This new mobile agent is same as the old mobile agent; the only difference being that they are in different agent contexts and environments.

When a mobile agent decides to migrate, it calls the move_to method in the SMART API either directly or indirectly. This method then hands this agent over to the agent context or the place server where the agent is currently executing. Note that the Proxy makes an RMI call to the place server, because place server has been implemented as an RMI server. The Proxy is a Java object and it has to pass the reference of itself to the place server. Since CORBA allows only IDL types to be passed for remote function calls, implementing the place server as a CORBA server will not allow the Proxy object to be passed to the place server as the Proxy object is not an IDL type. On the other hand, RMI allows Java objects to be passed to remote functions. Hence the place server has been implemented as an RMI server. When the agent starts up, the default place name is "A1". This can also be specifically overridden by specifying this information in the agent configuration file. The agent context/place then serializes the agent object by object serialization techniques and converts it to a sequence of bytes. It then passes this sequence of bytes while calling receive_agent in the target agent system. This target agent system finds the place server where this agent should go to and then deserializes this agent and hands over this agent to the destination place server, which then starts the mobile agent as a thread. For this reason, all the mobile agents programmed in SMART have to implement the *run*( ) method in their code. This is the method, which will be called every time the agent migrates to a remote target and starts as a new thread. The resultant agent thread, which is started in the destination place is same as the original agent which left the source place. Hence the agent in effect, has been migrated by SMART.

4.1.3. Implementation of dynamic aggregation

As discussed in our design, only the mobile agents will have access to the dynamic aggregation features, as our primary reason for developing this facility is to make transfer of agents more efficient. The base class for all mobile agents will incorporate all the code to support dynamic aggregation but some of the related code will have to be incorporated in place server. This is because the place server is responsible for serializing the agent and during that process it can detach those facets of the agent that cannot be. Place server is also responsible for maintaining remote references of the left out facets with the agent.

Internally dynamic aggregation is implemented by a set of hashtables and some supporting methods. The hashtables maintain the references to the facet objects and the supporting methods can be used by the mobile agent to manage the facets. These methods allow an agent to attach a facet in different modes if it already has a reference to the object (to be attached as a facet) or its class information. It can also detach the facets, query about facets and avoid duplicating facets.

A facet can be attached in the following modes:

- *Search mode*: In this mode, the class information provided by the agent is combined with the class information of the agent itself to create and attach the most suitable facet. For example, an agent of class "Mechanic" wants to repair a vehicle of type "Car" then the most appropriate facet to be attached will be "CarMechanic".
- *Sticky mode*: In this mode, the agent has the option of keeping the facet attached to itself when it moves from one host to another. When this option is turned off for a facet, the agent is stripped off of that facet before it leaves a place server to go to another place server.
- *Remote mode*: This option works only when the sticky mode of the facet is turned off. With this, the agent has the option of maintaining a remote reference to the left out facets (non-sticky facets).

4.1.4. Network class loading

Network class loading is a feature by which the place server can receive an agent even if it does not have the class information of the agent for deserialization. This is an important feature for a mobile agent platform as it relieves the burden of pre-installing the agent and other related class information on all the sites the agent is going to visit. Sometimes it may not be possible to determine the itinerary of the agent before hand.

The code for the network class loaders is installed as part of the place server because the place server is responsible for deserializing an agent. If the place does not find the agent class information for deserialization it invokes the network class loader to get the class information from the place server where the agent was created. The network class loader downloads the agent's class information and defines the class. In the process of defining a class from the class information, the network class loader may come across some more classes whose information is not available in the local classpath. In such cases, the network class loader recursively applies the process of downloading class information and defines them.

4.1.5. Tracking the agent

Voyager provides the feature of tracking the agent by holding a remote reference to the agent after the agent leaves a particular site. For example, if an agent leaves site S1 and goes to site S2, Voyager provides the API for S1 to hold a remote reference of the agent which is now executing on S2. This essentially becomes equivalent to client–server paradigm. With such features available, the programmers will be tempted to develop mobile applications that use remote agent references extensively and hence defeat the whole concept of using mobile agents. A well designed mobile agent application will not need remote agent references (to save network bandwidth). In fact, a good mobile agent application should allow the user to log out or stop consuming the CPU once the agent has been launched. The agent, upon its return, logs itself with the originating agent system and is held there until the user logs in and checks the agents status.

We have implemented a similar agent tracking system in SMART. The agent system where the agent originated is not allowed to hold a remote reference to the agent once the agent is launched. After finishing the trip, the agent registers itself with the originating agent system. It is to be remembered that the agent system keeps running even if the user logs out of the system. The user can log in anytime later and run a program that can acquire a reference to the parked agent by giving the agent-key to the agent system. Agent-key is an agent identifier that the user gives to the agent and must be remembered by the user.

Fig. 2 shows the functionality of various components in SMART and how they communicate with each other.

4.2. Class description

Table 2 gives a brief description of important classes in SMART, their super classes, the place where they fit in and their functionality.

4.2.1. Smart API

For creating a mobile agent application the following API is available to the user:

- *MobileAgent*( ): This must be called from the agent's constructor. This initializes the agent's itinerary and configures various parameters.
- *void create_instance*( ): Creates a local or a remote instance in a specified target host.
- *void move_to*( ): Moves the agent to the next host that is specified in its itinerary. The agent starts running from its run( ) method once it is resurrected on the destination host.

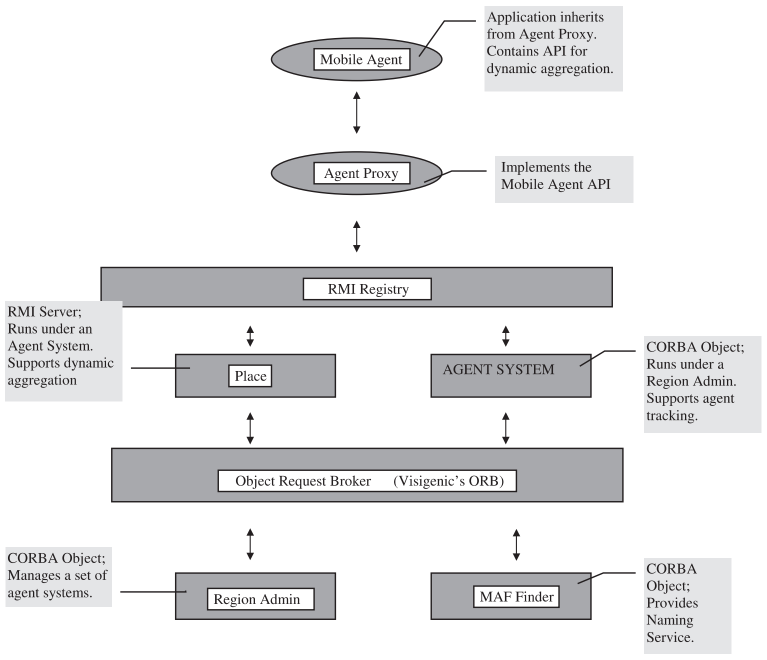
Mobile Agent
Application inherits from Agent Proxy. Contains API for dynamic aggregation.
Agent Proxy
Implements the Mobile Agent API
RMI Registry
RMI Server; Runs under an Agent System. Supports dynamic aggregation
Place
AGENT SYSTEM
CORBA Object; Runs under a Region Admin. Supports agent tracking.
Object Request Broker (Visigenic's ORB)
CORBA Object; Manages a set of agent systems.
Region Admin
MAF Finder
CORBA Object; Provides Naming Service.

Fig. 2. Communication between SMART components.

Table 2
Description of classes in SMART

Class name	Super class/interface	Layer	Functionality
Mobile agent	Extends thread implements serializable	Agent proxy	Contains the mobile agent and dynamic aggregation API
AgentConfigReader	Extends object	Agent proxy	Extracts information about the agent from its configuration file
PlaceImpl	Extends uni cast remote object implements place, serializable	Place	Provides context for agent execution. Runs agent as a thread. Contains network class loader
Place_AgentTable	Extends object	Place	Table maintained by place to keep track of its agents
AgentSystemImpl	Implements agent system	Agent system	World of agents. Allows agents to be created, moved, executed, suspended and terminated. Holds the agent after the agent trip is over
AgentSystem_AgentTable	Extends object	Agent system	Table maintained by the agent system to keep track of its agents
AgentSystem_PlaceTable	Extends object	Agent system	Table to keep track of places in the agent system
AgentSystemConfigReader	Extends object	Agent system	Extracts information about the agent system from its configuration file
RegionImpl	Implements region	Region	Controls all the agent system with the same authority
Region_AgentSystemTable	Extends object	Region	Table maintained by the region administrator to keep track of its agent systems
FinderImpl	Implements Finder	Region	Provides naming service for a region
AgentRegistrationInfo	Extends object	Region	This information is used by the Finder to keep track of the agents in its region
AgentSystemRegistrationInfo	Extends object	Region	This information is used by the Finder to keep track of the agent systems in its region
PlaceRegistrationInfo	Extends object	Region	This information is used by the Finder to keep track of the places in its region

- *void get_agent_profile*( ): Allows the agent to query about its profile.
- *set_authinfo*( ): Allows the agent to set or change its authentication information.

4.2.2. Dynamic aggregation API

From a user's perspective the following dynamic aggregation API is available as part of the MobileAgent class:

- *boolean addFacet*(*Object object*, *boolean sticky*, *boolean keepalive*): Adds an object as a facet to the agent. If this object is already a facet then nothing will happen and false is returned. A null object will not make any changes to the facet list of the primary object. Returns true if the facet is added successfully. The sticky flag will let the place server know if this facet object has to be transported along with the agent when the agent moves. The keepalive flag works only if the sticky flag is set to false. The facet object will not move with the agent if the sticky flag is false and the keepalive flag will tell if the agent wants to maintain a remote reference to this facet. Hence forth, the sticky and the keepalive flags will hold the same meaning in the other APIs as well.
- *Object createFacet*(*String string*, *boolean search*, *boolean sticky*, *boolean keepalive*): This will create a facet that implements the specified class (string). The rules for finding a suitable class depend on the search flag. If the search flag is false then the string is used as the name of the class else the string is appended to the class of the agent and that becomes the name of the class. If such a facet exists which implements the specified class then a reference to that facet is returned else a new facet is created and its reference is returned.
- *boolean deleteFacet*(*Object object*): Removes the object as a facet of the primary object. If the object does not exist as a facet then nothing will happen and a false is returned.
- *boolean deleteFacetByClass*(*String string*, *boolean search*): Deletes the facet of the primary object that implements the specified class. The rules for finding the appropriate class of the facet are the same as described above. If no such facet is found then false is returned.
- *void deleteAllFacets*( ): Removes all the facets of the primary object. If no facets exist then nothing will happen.
- *void deleteAllFacetsByClass*(*String string*, *boolean search*): Removes all the facets of the primary object that implement the specified class. The rules for finding such a class are the same as described above.
- *Vector getFacets*( ): Returns a vector containing all facets of the primary object.
- *boolean isFacet*(*Object object*): Returns true if the object already exists as a facet to the primary object.

4.3. Testing

4.3.1. General testing

Each component in SMART was tested as and when it was developed along with the other existing components. The entire project has been designed and implemented using object oriented techniques. After the Place, Region, MAFAgentSystem and MAFFinder were designed, the MAFFinder was first implemented and tested alone. Then the Region was built and tested thoroughly. The Region was then integrated with the MAFFinder and tested. The AgentSystem was then implemented and tested along with the MAFFinder and Region. Both black box and white box testing techniques were used while testing the system in various situations. Place was implemented on top of the thoroughly tested agent system and further testing was carried out by integrating all the above modules. Some trivial and non-trivial mobile agent applications were built on SMART so that all the main execution paths of the above modules would be reached. These applications were then used to test the whole system thoroughly. These applications involve considerable amount of concurrency and mobility. Separate applications were developed to test dynamic aggregation, network class loading and agent tracking facilities in SMART. We also ported two applications that were developed on Voyager to SMART. The ported applications gave exactly the same results as the applications running on Voyager. The current SMART system has withstood all the above testing and has proved to be a reasonably robust and efficient system fulfilling our design goals. Two of those applications which were used for the above-mentioned testing, will be described in the next section in detail.

4.3.2. Examples for testing

In this section we will look at some of the testing applications that were used to test various features of SMART.

Testing dynamic aggregation: We developed a car mechanic application in which the agent is a car mechanic and it would move from one host to another repairing cars. Since there are numerous types of cars available, it would be very inefficient for the agent to carry all manuals and tools to repair all possible types of cars available on this planet. Instead, the agent uses dynamic aggregation feature to attach required manuals and tools depending on the type of the car it is going to repair. Upon reaching a site where the agent has to repair a car, the agent determines the type of the car and loads appropriate facets. In our example, we also dealt with the situation when the local host does not have class information of the facet the agent wants to attach. In that case, the facet class information is obtained from the originating place server using network class loaders. Once the work is done, the agent detaches the facets to reduce its size and launches itself to the next site.

Testing network class loading and agent tracking: In this application we created a scenario in which one of the agent systems which the agent is going to visit does not have any class information about the agent. When the agent reaches such an agent system, an exception is raised during the deserialization of the agent. Then the agent system uses the network class loader to get the class information from the originating place server. In this application we also tested the agent tracking system.

Porting Voyager applications to SMART: In this testing procedure, we ported two applications developed on Voyager to SMART. The purpose of this test was to see if SMART is lacking in any major requirements, which a programmer would need for developing mobile agent applications. We also wanted to compare the results obtained from the ported applications with those obtained from the applications running on Voyager. The results of this testing were as expected. We could port the applications without many difficulties. Note that the applications running on Voyager did not use any advanced features of Voyager such as object persistence, multicast messaging, mobile directory services and agent security.

5. Conclusion

5.1. Summary

Mobile agent programming is a new and fast developing paradigm for distributed programming. This paradigm has a lot of potential for building new and fascinating distributed intelligent applications in the fields of electronic commerce, data mining and other intelligent application areas. In most of these cases, it can out-perform the traditional client–server approach. Recent studies have shown a need for compliance between agents developed on various platforms and in heterogeneous environments. For this purpose a large consortium of research organizations and companies has developed a specification known as Mobile Agent Facility (MAF). Currently, the MAF specification is at its infancy and still a lot of refinements needed to make it a better and complete industrial standard. We believed that "learning by doing" is the best way to experiment with this new programming paradigm. Hence we designed and implemented this new MAF compliant workbench prototype, which focuses on mobility, standard compliance, efficiency and scalability. We named this system as Scalable, Mobile and Reliable Technology (SMART).

During the course of development of SMART, we have come across the need for several necessary and important features that are not defined in MAF specification. These features include dynamic aggregation and network class loading. We have implemented and incorporated these features in SMART without effecting its MAF compliance. We have also focused on making SMART an efficient and scalable system.

SMART has turned out to a very good working model of MAF specification and it provides an excellent platform for developing fairly complicated mobile agent applications. Since we have access to the source code, any future changes can be readily made and new ideas can be implemented and integrated with SMART easily. This is not possible with commercially available mobile agent platforms like Voyager and Aglets Workbench since their source code is not freely available.

5.2. Future work

The mobile agent technology, by all means, is still at its infancy. Many difficult issues that have to be resolved before the mobile agent applications become commercially viable. Most important among these are knowledge representation and network security. Some work has been done in both these domains. Knowledge Query and Manipulation Language (KQML) is both a message format and message handling protocol designed to support runtime knowledge sharing between agents. Only with the use of KQML can mobile agent systems reach the complexity level that will be demanded by commercial applications (Kinitry and Zimmerman, 1997). The open group is continuing to refine its specifications on security for multi-hop agents, which travel across multiple security domains. Security issues for such a situation are very complex and the open group will hopefully be able to come up with a new set of specifications for such situations. Other security areas that need further work are ensuring security for the agent and the information it is carrying.

Following are some of the fields in which SMART needs some work to be done. Although these fields are yet to be addressed in its entirety in the MAF specification, we can work ahead and look at developing these fields in SMART.

- *Object persistence and database support*: Object persistence will give unlimited lifetime to an agent which could be a very useful feature. To implement this, SMART would need a basic object store manager if not a full-fledged database support.
- *Transaction service*: Recently ObjectSpace has included transaction services as part of Voyager. The importance of a transaction service cannot be undermined in mobile agents especially those deployed in electronic commerce. Currently, SMART does not have any support for transaction management.
- *Security*: Security is the single most important issue in mobile agent systems. It is also the most complicated and puzzling issue to solve. Currently, SMART does not have any security policies. It trusts all agents and agent systems equally. MAF is continuing to refine its specifications on security and will hopefully come up with complete specifications in the near future.

SMART should implement these specifications, if it has to be commercialized.

- *Fault tolerance*: In distributed systems, as opposed to centralized systems, the point of failure can lie with any of the numerous components spread out on the network. A single failure anywhere in the system can cause the agent to vanish in thin air or lose vital information. Currently, SMART uses Java based exception handling system to deal with failures. This is not sufficient to provide good fault tolerance. We need to integrate better-distributed fault tolerance algorithms in SMART.
- *Underlying ORB*: We can change the underlying CORBA ORB with the ORB that is available with Java 1.2 to make SMART an all Java product.

References

Bradshaw, J.M. (Ed.), 1997. Software Agents. The MIT press, Cambridgre, MA.

IBM Inc., 1997a. Fiji – running aglets on web pages. http://www.trl.ibm.co.jp/aglets/infrastructure/fiji/fiji. html.

IBM Inc., 1997b. Aglets – components. http://www.trl.ibm.co.jp/aglets/infrastructure/.

IBM Research, 1998. Subject oriented programming. http://www.research.ibm.com/sop/sophome.htm.

Javaworld. The basics of Java class loaders. http://www.javaworld.com/javaworld/jw-10-1996/jw-10-indepth. html.

Kiniry, J., Zimmerman, D., 1997. A hands on look at Java mobile agents, July–August 1997. Internet Computing Journal – IEEE.

Lange, D.B., 1996. Aglets documentation. http://www.trl.ibm.co.jp/aglets/documentation.html.

Mitsubishi Electric. Mobile agent computing using concordia. http://www.meitca.com/HSL/Projects/Concordia/MobileAgentsWhitePaper.html.

ObjectSpace Inc., 1997. Voyager project homepage. http://www.objectspace.com/voyager/.

OMG, 1997. The mobile agent facility specification – draft 7. http://www.genmagic.com/agent/MAF.

OMG, 1998. CORBA for beginners. http://www.omg.org/news/begin.htm.

Prentice-Hall. Object oriented software design and construction. http://www.prenhall. com/divisions/esm/app/kafura/secure/chapter4/html/4.7_dynamic.html.

Sun Inc., 1998. Distributed object computing. http://www.sun.com/sunworldonline/swol-04-1996/swol-04-oobook. html.

Sun Inc. The Java package index. http://java.sun.com/products/jdk/1.1/docs/api/packages.html.

Tham, C., 1996. Mobile agent facility conformance proposal, December. http://www.agent.org/pub/satp/papers/mafConformance.html.

University of Ottawa. Intelligent mobile agents research page. http://deneb.genie.uottawa.ca/.

Visigenic Inc., 1997. VisiBroker for Java and C++. http://www.visigenic.com/prod/vbrok/vb30DS.html.

White, J., 1996. Mobile agent whitepaper. http://www.genmagic.com/agents/Whitepaper/whitepaper.html.

White, T., 1998. Lecture on mobile agent facility, February. http://www.sce.carleton. ca/netmanage/mctoolkit/mctoolkit/lecture/tsld041.htm.

Johnny Wong is a full Professor of the Computer Science Department, Iowa State University at Ames, Iowa, USA. His research interests include Operating Systems, Distributed Systems, Telecommunication Networks, Broadband-Integrated Services Digital Networks, Concurrency Control and Recovery, Multimedia and Hypermedia Systems, Intelligent Multi-Agents Systems, Intrusion Detection. He has been an investigator fo research contracts with Telecom Australia from 1983 to 1986, studying the performance of network protocols of the ISDN. During this period, he has contributed to the study and evaluation of the communication architecture and protocols of ISDN. From 1989 to 1990, he was the Principal Investigator for a research contract with Microwave Systems Corporation at Des Moines, Iowa. This involved the study of Coordinated Multimedia Communication in ISDN. In summers 1991 and 1992, Dr. Wong was supported by IBM corporation in Rochester. While at IBM, he worked on the Distributed Computing Environment (DCE) for the Application Systems. This involved the integration of communication protocols and distributed database concepts. Dr. Wong is also involved in the Coordinated Multimedia System (COMS) in Courseware Matrix Software Project, funded by NSF Synthesis Coalition Project to enhance engineering education. From 1993 to 1996, he is working on a research project on a knowledge-based system for energy conservation education using multimedia communication technology, funded by the Iowa Energy Center. From 1995 to 1996, he was supported by the Ames Laboratory of the Department of Energy (DOE), working in Middleware for Multidatabases system. Currently, he is working on Intelligent Multi-Agents for Instrusion Detection and Countermeasures funded the Department of Defense (DoD), Database Generating and X-Ray Displaying on the World Wide Web Applications funded by Mayo Foundation and CISE Educational Innovation: Integrated Security Curricular Modules funded by the National Science Foundation (NSF).

Guy Helmer is a Ph.D. candidate in Computer Science at Iowa State University, researching security in operating systems and networks. Guy received his B.S. degree from the South Dakota School of Mines and Technology in 1989 and his M.S. degree from ISU in 1998. Guy served for several years as a system programmer and network engineer for Dakota State University. During that time he designed and maintained local and wide area networks, administered UNIX, Netware, and Windows NT servers, and consulted with state government agencies and other universities on networking and security issues.

Venky Naganathan received his B.E. degree in Computer Science and Engineering in May 1996 from Anna University, India, and the M.S. degree in Computer Science in August 1998 from Iowa State University. Currently, he is working as a software engineer at Hewlett-Packard. His reserach interests include distributed application architecture, object-oriented modeling, Internet protocols/systems architecture.

Sriniwas Polavarapu received Bachelor of Technology degree in Computer Science and Engineering in June 1996 from Jawaharlal Nehru Technological University, Hyderabad, India and M.S. degree in Computer Science in August 1999 from Iowa State University. He has also done Masters studies in Computer Science at the Indian Institute of Technology, Kanpur, India. Currently, he is working as a member of technical staff at the AT&T Labs., San Jose, California. His research interests include scalable client–server technology, mobile agent systems and distributed and object-oriented databases.

Vasant Honavar received his B.E. in Electronics Engineering from Bangalore University, India, and M.S. in Electrical and Computer Engineering from Drexel University, and an M. S. and a Ph.D. in Computer Science from the Univeristy of Wisconsin, Madison. He founded and directs the Artificial Intelligence Research Laboratory (www.cs.iastate.edu/~honavar/aigroup.html) in the Department of Computer Science at Iowa State University (ISU) where he is currently an associate professor. Honavar is also a member of the Lawrence E. Baker Center for Bioinformatics and Biological Statistics, the Virtual Reality Application Center, and the faculty of Bioinformatics and Computational Biology at ISU. His research and teaching interests include Artificial Intelligence, Machine Learning, Bioinformatics and Computational Biology, Grammatical Inference, Intelligent Agents and Multi-agent systems, Distributed Intelligent Information Networks, Intrusion Detection, Neural and Evolutionary Computing, Data Mining, Knowledge Discovery and Visualization, Knowledge-based Systems, and Applied Artificial Intelligence. He has published over 90 research articles in refereed journals, conferences and books, and has co-edited three books. He is a co-editor-in-chief of the Journal of Cognitive Systems Research published by Elsevier. His research has been partially funded by grants from the National Science Foundation, the National Security Agency, the Defense Advanced Research Projects Agency (DARPA), the US Department of Energy, the John Deere Foundation, the Carver Foundation, Pioneer Hi- Bred Inc., and

IBM. Prof. Honavar is a member of ACM, AAAI, IEEE, and the New York Academy of Sciences.

Les Miller is a professor and chair of Computer Science at Iowa State University. His research interests include databases, data warehouses, integration of heterogeneous distributed data sources, and mobile agents. He was the editor of ISMM's International Journal of Microcomputer Applications from 1989 to 1999. He has also served as the vice-president of the International Society for Computers and their Applications (ISCA).