A Survey of Mobile Agent Systems

by

Syed Adnan
John Datuin
Pavana Yalamanchili

CSE 221
Final Project
June 13, 2000

Introduction

As the Internet constantly expands, the amount of available on-line information expands as well [3]. The issue of how to efficiently find, gather, and retrieve this information has led to the research and development of systems and tools that attempt to provide a solution to this problem. These systems and tools are based on the use of mobile agents.

Mobile agents are processes (i.e., executing programs) that can migrate from one machine of a system to another machine (usually in the same system) in order to satisfy requests made by their clients [2]. They implement a computational metaphor that is analogous to how most people conduct business in their daily lives: visit a place, use a service, and then move on [2]. Basically, a mobile agent executes on a machine that hopefully provides the resource or service that it needs to perform its job. If the machine does not contain the needed resource/service, or if the mobile agent requires a different resource/service on another machine, the state information of the mobile agent is somehow saved, transfer of the mobile agent to the machine containing the necessary resource/service is initiated, and the mobile agent resumes execution at the new machine. Advantages of using mobile agents include low network bandwidth since they only move when they need to move, continued execution even when disconnected from the network, ability to clone itself to perform parallel execution, easy implementation and deployment, and reliability.

Mobile agents have been developed as an extension to and replacement of the client-server model [4]. In the client-server model, a server is a machine that provides some service (or set of services) and a client (most often another machine) makes requests for those services. Communication between a client and a server is usually through message passing. So, when a client needs a particular service, it usually sends a request message to a server that contains the needed service. A limitation of the client-server model is that the client is limited to the operations provided at the server [4]. So, if a client needs a service that a particular server does not provide, the client must find a server that can satisfy the request by sending out messages to all servers. This clearly is an inefficient use of network bandwidth. Also, this severely limits network scalability since managing and updating these servers would prove prohibitive.

In this paper, we will survey several mobile agent systems. A mobile agent system provides the execution environment for mobile agents [1]. Sometimes called middleware, mobile agents systems also provide a framework in which mobile agent applications can be developed and managed.

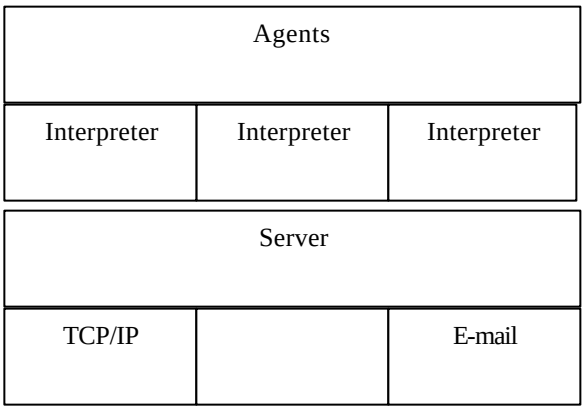
Mobile Agent Systems

In this section, we will take a look at six mobile agent systems: Agent TCL, ARA, Concordia, Mole, Tacoma, and Voyager. Though these systems differ in their goals, motivations, and implementations, they all (more or less) provide common functionalities that support: the migration of agents, the communication between agents, various programming/interpreted languages, and various forms of security.

Agent TCL/D'Agents

Agent TCL (later renamed to D'Agents) is a mobile agent system created at Dartmouth College to address the weaknesses of existing mobile agent systems, such as insufficient security mechanisms, support for only specific and complex languages, difficult or nonexistent communication between agents, and inadequate migration facilities [4]. The architecture of Agent TCL is based on the server model of Telescript and supports a modified version of the Tool Command Language (TCL) as its high-level scripting language implementation [4] (support for Java and Scheme is being added in D'Agents [5]).

The architecture of Agent TCL consists of four levels:



The lowest level contains each supported transport mechanism. The next level is the server level that manages local and incoming agents. A server that runs at each machine of the network site performs such tasks as:

- Keeping track of agents local on its machine
- Accept and authenticate incoming agents
- Provide a hierarchical namespace for each agent and service
- Allow agent communication via messages
- Allow agent migration
- Provide access to a non-volatile store so agents can save and restore their internal state as desired and in the event of a node failure, respectively [4]

The interpreter level provides the execution environments for each supported agent language [5]. So, if an agent is written in TCL, the server hosting the TCL-written agent loads the TCL interpreter to allow the agent to execute; similarly, the Java virtual machine is used for agents written in Java. The last level is the agent level that contains the agents themselves. The agents execute in the interpreters and use the facilities provided by the server to migrate from machine to machine and to communicate with other agents [5]. There are two types of agents: those that move from machine to machine accessing resources, and those that remain on the machine and whose purpose are to provide specific services not inherently provided by the system (e.g., navigation, high-level communication protocols, and resource management) [5].

Agents migrate using the *agent_jump* command. The modified TCL language provides this command, as well as others that will be discussed later. The method used in moving agents is determined by the transport mechanisms supported by the server the agent is migrating from (e.g., TCP/IP). The agent_jump command captures the internal state of the agent and sends this information to the destination machine. The server on the destination machine loads the appropriate interpreter for the agent, restores the migrated agent's state information into this execution environment, and resumes the agent's execution at the statement immediately after the agent_jump [4]. The agent is now on the destination machine and can interact with that machine's resources without any further network communication [5].

Agent TCL also provides the simple commands *agent_meet*, *agent_accept*, *agent_send*, and *agent_receive*. *Agent_meet* and *agent_accept* are used to establish a direct connection between agents [5]. This connection is used for communication purposes. When a source agent wishes to communicate with a destination agent, the source agent issues the *agent_meet* command that is sent to the destination agent. The destination agent uses the *agent_accept* command to receive the *agent_meet* command and can either accept or reject the request from the source agent. If the request is accepted, the source agent is informed of the port in which to connect to begin communication with the destination agent [4]. The *agent_send* and *agent_receive* commands are used to facilitate the sending of messages between the agents.

Security in Agent TCL is provided in various capacities. To protect migrating agents and to provide authentication (e.g., to verify the identity of an agent's owner), Agent TCL uses Pretty Good Privacy (PGP) for its digital signatures and encryption [5]. To protect resources, a resource manager assigns each agent a set of access permissions [5]. So, when an agent tries to access a resource, the request is sent to the resource manager that checks the agent's access permissions with the resource. If the agent does not have the proper permission, it is denied access to the resource. To prevent agents from performing malicious acts, each interpreter is extended to include a security module that prevents such acts (e.g., forging a pointer to try to gain access to unauthorized data).

Agent TCL has been used in both information-management and information retrieval applications [4].

ARA

Ara is a platform for the portable and secures execution of mobile agents in heterogeneous networks. Mobile agents in this sense are programs with the ability to change their host machine during execution while preserving their internal state. This enables them to handle interactions locally which otherwise had to be performed remotely. Ara's specific aim in comparison to similar platforms is to provide full mobile agent functionality while retaining as much as possible of established programming models and languages. Mobility should be integrated as comfortably and unintrusively as possible with existing programming concepts-algorithms, languages, and programs [9].

A mobile agent in Ara is a program able to move at its own choice and without interfering with its execution, utilizing various established programming languages and the platform provides facilities for access to system resources

and agent communication under the characteristic security and portability requirements for mobile agents in heterogeneous networks. Portability is an issue because mobile agents should be able to move in heterogeneous networks to be really useful and security is important because the agent's host effectively hands over control to a foreign program of basically unknown effect. Most existing platforms do not run the agents on the real machine of processor, memory and operating system, but on some virtual one, usually an interpreter and a run-time system, which both hides the details of the host system architecture as well as confines the actions of the agents to that restricted environment. This is also the approach adopted in Ara – mobile agents are programmed in some interpreted language and executed within an interpreter for this language, using a special run-time system for agents, called the core in Ara terms [9].

Core is the central part of an Ara system, implementing the basic concepts such as agents, allowances, service points, migration etc [8]. The core for reasons of security and portability mediates any access from an application agent to the host system or to another agent. The core treats agent independently of their programming language, using assistance from the language interpreters for language specific tasks.

Ara is primarily concerned with system support for general mobile agents regarding secure and portable execution, and much less with application-level features of agents, such as agent cooperation patterns, intelligent behavior, user modeling etc. The application focus of Ara is on weak-connection/high-volume systems such as wirelessly or intermittently connected computers, or globally distributed large databases. Such environments with intrinsic restrictions regarding the ratio of bandwidth/connectivity vs. data volume seem particularly well suited for mobile agent applications.

The programming model of Ara consists of agents autonomously moving between and staying at places, where they use certain services, provided by the host or other agents, to do their job. A place is physically located on some host machine, and may impose specific security restrictions on the agents entering that place in the form of a local allowance limiting the agent's resource accesses while staying at that place [8]. Besides that, an agent may also be equipped with a global allowance by its principal, controlling the agent's behavior throughout its lifetime. Keeping this in mind, agents are programmed much like conventional programs in all other respects, i.e. they work with a file system, user interface and network interface.

The system offers a clear interface to adapt interpreters for established programming languages to the core, demonstrated by the adoption of interpreters for such diverse languages as C/C++ and Tcl. Ara offers full migration of agents, i.e. orthogonal to the conventional program execution, which relieves the programmer of all details involved with remote communication and state transfer. Ara agents can migrate at any point in their execution, simply by using a special core call, named *ara_go* in Ara's Tcl interface [9].

The security model of Ara is flexible in that domains of protected resources can be dynamically created in the form of places, and that the admission of agents to such a domain, as well as their actual rights at that place, can be controlled in a fine grained manner down to individual agents and resources.

However, the described architecture [8] is still lacking in the area of structured agent interoperation. Further, supportive services for distributed resource discovery will be needed for real world applications.

An important area of application for mobile agents in Ara is in mobile computing. The Ara software currently runs on various types of Unix operating systems (Sparc Solaris, Intel Linux, and Sparc SunOS).

Concordia

Concordia is a full-featured framework developed at Mitsubishi Electric Information Technology Center America's (MEITCA) Horizon Systems Laboratory [7]. It provides for the development and management of network-efficient mobile agent applications for accessing information anytime, anywhere, and on both wire-based and wireless device supporting Java [6]. The applications move around network machines running Concordia to access services such as databases and those provided by other agents.

At the highest level, a Concordia system consists of a Java Virtual Machine (JVM), a Concordia Server running on a machine in a network, and a mobile agent running in the system. Both the Concordia server and mobile agents are Java programs. The JVM is used for Concordia's runtime environment [7].

Concordia consists of a set of components that provides some type of service such as communication, security, persistent storage, administration, and so on. The component responsible for agent mobility is the Conduit Server. When an agent wants to initiate its transfer to another machine, it invokes the methods provided by the Conduit Server [6]. The Conduit Server will then suspend the agent and create a persistent image of it to be transferred [7]. The Conduit Server will inspect the agent's Itinerary (described below) to determine its destination, contacts the Conduit Server on the destination machine, and transfers the agent's image to the destination where it is again stored before being acknowledged and can resume execution [7].

A unique feature of Concordia's mobility mechanism is that it also provides for the transmission of state information detailing where the agent has been and what is has accomplished as well as where it is going and what it still has to do [6]. The notion of an *Itinerary*, which is a data structure stored and maintained outside of the agent itself, is used to describe an agent's travels [6]. Within the Itinerary is a list of *Destinations* that details the location (i.e., a hostname of a machine) where the agent is to travel to and the job (i.e., a method of an agent providing the requested service) it has to do. So, if a particular agent's itinerary consists of two locations (e.g., location A and location B) and two jobs (e.g., job A and job B), then the agent will first go to location A and perform job A, then travel to location B and perform job B; management of the agent's internal state is handled by the Concordia server. Concordia uses the Java Object Serialization (JOS) facility as the mechanism for the actual transfer of mobile agents [6]. When the Conduit Server transfers the agent, the agent is serialized into the format needed by the JOS facility, and deserialized at the destination machine.

Agent communication is either through asynchronous distributed events or collaboration. Asynchronous distributed events are events that agents receive via the Event Manager component. The agent determines the type of events an agent receives when it first registers with the Event Manager. The Event Manager can forward events to an agent even after it migrates to another system [6]. Besides sending events to each individual agent, Concordia provides the facility to send an event to a group of agents. Agents within an application that need to communicate or coordinate with each other do so via group-oriented events [6]. The only difference in the registration process is that an agent includes the group name to the Event Manager of the group that the agent belongs to. Collaboration extends agent communication by enabling multiple agents to perform complex distributed computations more effectively by correlating their results and altering their behavior based on the combined results [6].

Concordia's security model provides support for two types of protection: protection of agents from being tampered with, and protection of server resources from unauthorized access [6]. To protect agents during transfer, Concordia uses encryption (i.e., SSLv3). To protect resources on each server, Concordia relies on its Security Manager component to manage resource protection. Each agent is assigned an identity that is used when trying access resources. The Security Manager authenticates each agent by verifying its identity. If the identity matches, then the agent is able to access the resource. Concordia's resource protection is based on the user of the agent rather than the developer of the agent, as in other systems [6].

Mole

Mole is the first Mobile Agent System that has been developed in the Java language. The first version has been finished in 1995, and since then Mole has been constantly improved.
Mole provides a stable environment for the development and usage of mobile agents in the area of distributed applications.

In Mole system, agent model based on Agents and places. Each Agent's identifier is created at the creating of each agent, which uniquely identifies that agent globally. The philosophy of the system is that there are different kinds of mobility for mobile agents. There is Strong Migration and Weak Migration. In Strong Migration, the underlying system captures the underlying agent's entire state (execution state and data) and transfers it together with the code to a new location where the state of the agent is restored. This scheme is very attractive to programmers that it is transparent to the programmer, but it does have a high cost for system. On the other hand, Week Migration, which only transfer the data, state of the agent. The size of the transferred state information can be limited even more by letting the programmer select the variables making up the agent state. As a consequence, the programmer is responsible for encoding the agent's relevant execution states in program variables. Also, the programmer is required to provide a *start* method that decides, on the basis of the encoded state information, where to continue execution after migration. This method reduces substantially the amount of state to be transferred. But it changes the semantics of a migration, a fact that every agent programmer has to be aware of. Mole uses Week Migration scheme because one of the goals of Mole to run on any machine having a VM. And a normal java VM doesn't support capture of threads, which would have been required for the Strong Migration scheme. This scheme built in this system by using part of RMI package. When an agent calls a migrate method call to migrate, all threads belonging to that agent are suspended. After suspension of threads, agent is taken off from active agent list, and a (serialized) system independent representation of the object is created. Once agent is moved, object looks for all required classes and migrate the code from the code server if needed. Once object creates its thread on the destination, object sends a success message back to the sender system.

There are several different types of communication among agents. There is service to agent interaction, which is very much like a RPC type client/server communication. Second, mole has mobile agent communication among them, which use a concept called session, which is described in details later on the paper. There is anonyms group agent communication concept, and finally user agent communication. Most of the time there is either RPC or session based

communication among agents. Session based communication requires every agent to be identified by an identifier called 'badge'. Agents who wish to communicate must establish a session before any communication. After session setup, agents can communicate by remote methods or by message passing. Once all communions are completed among agents, session is terminated. There are two reasons for using session for communication among agents. First, concept of session can be used for synchronize the agent who wants to meet for cooperation. This concept was also introduced to allow agents to specify the type of agents they are interested and where they would like to meet. Secondly, the architect of Mole wanted to support 'stateless' and 'stateful' interactions. Either agent can terminate session.

Mole supports asynchronous communication by an even driven model. In event model, depending on these events, internal rules, state information and timeout intervals, output events are generated, that in turn may be the input for other synchronization objects. Mole allows synchronous and asynchronous messages among agents along with RPC type communication.

Mole uses a 'Sendbox' security model. In this model, service agents are agents with access to system re-source, providing controlled, secure abstractions of these resources inside the agent system. Furthermore, service agents may offer access to legacy software, using the native code interface offered by Java. This does not cause any security problems, because the service agents are immobile and may be started only by the administrator of the location. User agents may only communicate with other agents and have no direct access to system resources. Additionally it can be decided on a per-location basis which types of agents to allow on a place. Only agents that are derived from the specific type given can migrate to a place. This mechanism can be used to implement access restrictions.

Tacoma

An *agent* in TACOMA is a piece of code that can be installed and executed on a remote computer. Such an *agent* may explicitly migrate to other hosts in the network during execution. The TACOMA project focuses on operating system support for agents and how agents can be used to solve problems traditionally addressed by other distributed computing paradigms, e.g. the client/server model. An agent needs to store code and data for future computations. It must be able to carry this information around when it migrates, and later retrieves it. Also, agents should be allowed to leave data behind at hosts or share data with other agents. A folder represents this type of information in the TACOMA system. TACOMA agents store data in folders. A subset of the folders are identified with individual hosts and collected in the file cabinets managed by the hosts, the remaining folders comprise a briefcase that is moved from host to host along with the computation. Folders are organized in briefcases or cabinets [10]. The former is intended for violative and movable data, and the latter is intended for persistent and shared data. If a folder does not exist, it is automatically created when data is stored into it. Having different routines for briefcases and cabinets has advantages. Programs become more readable, the type of destination is visible by the name of the primitive used to access it. It also avoids semantic errors in which the application programmer stores folders in a briefcase when a cabinet was intended or the other way around. The alternative is to overload primitives for briefcases and cabinets, letting the library routine discover the type of destination intended. This reduces the number of primitives, which has to be maintained and documented. Fewer primitives make the API simpler.

A TACOMA agent executing on one host moves to another host by using TCP to communicate with TACOMA software at the destination host. TACOMA agents are migrated using a simple primitive called *meet* [10]. A TACOMA agent can cause another agent to be executed by invoking the *meet* operation and naming a target agent and a briefcase. The effect of the operation is to terminate the agent invoking the *meet* and then start executing the target agent with the specified briefcase. Thus, transfer of control in TACOMA from one agent to another is similar to the transfer of control enabled by using continuations in Lisp-like languages. Service agents are passively waiting to be activated by a *meet*. This is somewhat equivalent with the server blocking while waiting for an incoming request in the client-server model. In its simplest form, this delivery can be viewed as a procedure call. The folders of the briefcase are equivalent to the arguments of a procedure call, and the agent receiving that briefcase is equivalent to a procedure.

meet does not support automatic state capture or preemptive migration. However, the *meet* semantics now includes remote and local activation of service agents. And this activation can be synchronous, blocking the client agent until the service agent returns a briefcase or it can also be asynchronous, which blocks the client agent until the briefcase has been successfully delivered [10].

The receiving end of remote *meet*, is the bridgehead, which consists of the firewall that is the entry point to a host. Other entities include guardian processes, a cryptographic service agent, and the individual code service agents. Logging approach [10] to fault-tolerance survives a single host crash with no upper bound on recovery time.

Distributed applications have already been implemented using TACOMA to gather and visualize Arctic weather data to provide matching between service providers and potential clients, to communicate and interact with users (i.e., active documents), and to manage software installation in a network [11].

Voyager

Voyager is 100% java agent-enhanced Object Request Broker (ORB) created by ObjectSpace Company. Goals of this product to provide programmer to create state of the art distributed programs quickly, and easily while providing a lot of flexibility and extensibility for the products that are being created with the voyager system. Voyager supports RMI, DCOM, and CORBA architecture to provide stationary client server applications, which makes this system very flexible. This is a 100% pure java based system. Voyager uses regular java syntax to create remote objects and move them between applications. It transparently locates the agents and sends them message as they work, even if the agents are moving, all this is done for programmer. One of the great advantages of this system is that it supports both traditional client server architecture and agent-based architecture.

Objects are the basic building blocks of the voyager system, these objects resides and execute in voyager application. These objects reside in the voyager application, and every application is responsible for type of infrastructure these objects will use for remote communication, movement, and other support services. Any voyager application spawns a thread when it starts, and that thread takes care of timing facility, garbage collection, and manages TCP/IP message traffic. Every application in voyager system consist of its host and communication port an integer number that is unique to the host.

Agents are special type of objects in voyager applications. These objects are simply remote objects. These objects can exist outside of the local application's address space. Application communicates with remote object by creating a virtual version of the remote object. This virtual object acts as a reference of the remote object hiding the location of the remote object from the programmer. When messages are being sent to the remote object, virtual object forward the message to the remote object. If message requires a return value, virtual object receive the return value, and forward back to the caller of the original message. Using these virtual objects, several tasks can be performed besides sending messaging the remote objects, which are as fallows.

1) Remote object creation.
2) Connection with existing remote object in different applications.
3) Allows to move code and objects to another clients

Remote Object Creation:

When constructing remote object using virtual object, virtual object constructer will require the address of the remote object, if object is not there, then voyager's internals will move the code to the remote object, and create the object. Once remote objects are created 16-bit GUID is assigned to these objects, which uniquely identifies these objects.

Sending Message to Remote Object:

As mention earlier, virtual object forwards and receives the message from the remote object. System keeps track of exceptions in remote system, and raises those exceptions locally if they occur remotely because of messages.

Connecting to Remote Objects:

Virtual object requires address of the application for creating (if necessary) and connecting to the remote object of that particular application. Locating the objects and other details are transparent to the programmer.

Object Mobility:

A simple method call moves move object. Again virtual object would require the address of the destination application as parameter. Object waists till all pending messages are finish, and then move to specified location. Object does leave a secretary object behind which forwards messages to it.

Sending Messages to Remote Objects:

Virtual object keeps track of moved object by their last known address. If remote object moved from its last position, it will leave a secretary behind to forward the messages to its new location. Messages are returned using the same mechanism, but they contain the new location of the remote objects. After messages are being returned unnecessary secretaries are removed. I find that this method of communicating could be very expensive that if there are movements of objects from one system to many other systems, it could cause a lot of overhead and delay in messages to arrive to its proper location.

There is different type of message support for voyager applications. Messages are synchronous, and they block till return. However, one-way message are allowed who doesn't wait for a return value, and move on with the code.

Garbage collection is a service which is provided by the system, which uses a 'ping' remote objects and by looking at their life span, their life, and their activity decides to garbage collect them or not.

Transaction support is given by the two-phase commit, replication is not being offered in this system as well as it should have. Voyager system includes a flexible security framework, lightweight security implementation, support for secure network communications via SSL adapters, and firewall tunneling using HTTP or the industry-standard SOCKS protocol. *Requires Voyager ORB Professional.*

Comparison

From the previous section, it is apparent that there are differences among the various systems. The table below highlights the pros/cons of each system.

MOBILE AGENT SYSTEMS	PROS	CONS
Agent TCL/D'Agents	- Good support for migration - Simple communication facilities - Good security mechanisms	- Lacks execution constraints - TCL is inefficient, 10000x slower than optimized C, not object-oriented, provides no code modularization so difficult to write and debug large scripts, no facilities for capturing the internal state of an executing script [4]
ARA	- Can run agents concurrently using a fast thread package - Can clone themselves, duplicating their internal state - Migration can be routed over a wireless link - Core offers "service points" for agent interaction - Agents can checkpoint their internal state to disk for later restoration	- Lacks sufficient security mechanisms (e.g. authentication, access permissions) - Need better error handling for agents failing remotely - Only "CPU time" and "memory" resources covered by allowances - Mobile agents transmitted using TCP only
Concordia	- Flexible agent mobility - Support for agent interaction through collaboration - Support for agent persistence and recovery - Good security mechanisms - Reliable guarantee of agent transfer	- See below
Mole	- Run on any java VM - Good security - Allows good communication among Agents with the concept sessions.	- Lacks synchronous communication and execution constraints - Only support java - Does not support strong migration - Coding is not easy
Tacoma	- Rear guard agents - Electronic cash - Broker agents - Generality of folder and *meet* mechanisms decouples Tacoma from the choice of language used in writing individual agents - Agent left in a well defined state and problems of blocking kernel	- Requires programmer to explicitly capture state information before migration [4] - Rudimentary security mechanisms - Software-installation problems. - Agents cannot be preempted by the TACOMA system, only stopped or aborted. - Pure binary data not very well

	calls is avoided because agent decides when it should migrate	supported by Tacoma
Voyager	- 100% java - No special syntax (no IDL) - Simple communication - Good support for transaction (2 phase commit) - Support for RMI, DCOM, CORBA	- Only support java - Not so great messaging system

Table 1 - Mobile Agent Systems Pros and Cons

There is obviously room for improvement in each system. Below, we will discuss the limitations, issues, and problems encountered by the design group of each system, as well as our views based on our observations.

Agent TCL

The version of Agent TCL primarily discussed in this paper only supported TCL (a later version, named D'Agents, just started to introduce support for Java and Scheme). The designers of TCL discovered some drawbacks in the use of TCL as the system's language support:

1) It is inefficient compared to other interpreted languages and is ten thousand times slower than optimized C
2) Its non-object-oriented nature provided no code modularization aside from procedures, thus making it difficult to write and debug large scripts (since TCL programs are really scripts)
3) It lacked facilities for capturing the internal state of an executing script, which are essential for providing transparent migration at arbitrary points [4]

Another limitation in the system was in its lack of constraining agent execution. Resources, directly accessible via TCL, are freely available for use by agents. Protection for resources not directly accessible via the language support for TCL is provided by third-party software [5]; the Agent TCL system itself does not inherently provide a means to prevent agents from doing something malicious if due to a programming error.

Even with these drawbacks, Agent TCL appears to be a decent system. With its support for Java, creating mobile agents will be easier for a lot of programmers. Also, with its simple communication facilities (e.g., agent_jump, agent_send, and agent_receive), layered architecture, and use of third-party software (e.g., PGP), Agent TCL systems are much easier to manage, enhance, and extend.

Issues lacking sufficient detail or missing entirely in looking at Agent TCL are how do agents discover resources that are available in a system, what happens in case of a system failure, how long does an agent live if it's owner disconnects from the system for a long time, and how does the underlying OS play a role in the system. This last issue dealing with the OS is particularly interesting. It appears that in these papers (and those of the other systems, with the exception of Tacoma), the OS is the "catch all" for things not supported by the system. Protection, communication, agent creation/execution/deletion are handled by the system (or more exactly the language, interpreter, virtual machine). There isn't any explicit mention of the OS, but things must have been leveraged on the OS since the system runs on top of the OS of the machine. Hopefully, in future papers on mobile agent systems, these issues will be discussed.

ARA

Some of the work in progress for Ara with regard to issues pointed out in the pros/cons section is as follows:

- Adapting Java to Ara to make it usable for mobile agent programming
- Multiple, user-defined places with security policies programmable by application code in the form of issuing individual local allowances to entering agents
- Make all the important resources subject to allowances: Files, network connections, disk space, network bandwidth, visited places
- Agent transmission by SMTP (i.e. e-mail), in addition to TCP

- Access to files and network connections to be scheduled and authorized by the core (currently direct calls to the host operating system are used)
- Encryption and authentication of migrating agents
- Unreliable asynchronous inter-agent messaging
- Better error handling for agents failing remotely, etc.

It wasn't clear from the papers on Ara, how certain issues like creating compiled agents at run time, garbage collection (if any) is done and how the system performs with respect to some application. Also, as stated before, supportive services for distributed resource discovery will be needed for real world applications.

Concordia

It was a little difficult finding a fault with this system since the papers did a good job on focusing on Concordia's strengths. The many components of the system (e.g., Persistence Manager, Conduit Server, Security Manager) provide a wealth of services and support to its mobile agents; much more than the other systems. The use of persistent storage to store the agent's information during and after transfer ensures that an agent is reliably transmitted to its destination. The added benefit of its use of persistent storage is in recovery. If the system goes down, agents can be restored when the system is brought back up. The Itinerary/Destination model provides a simple mechanism for defining and tracking how an agent travels, which allows agents to modify their Itineraries at runtime [6]; in other systems, its difficult to determine where the agent is going without explicitly looking at its code. Security is another strength. Unlike other systems where security is based on what agent developer decides, Concordia bases security on the rights of the user of the application, leading for more control of which files, databases, and resources are allowed for each user [7].

Of course, a lot of details were missing in the papers, such as how agents find out available resources in the system (though it appears the Service Naming component contained a list of applications and agents, but how that list is created/managed isn't apparent), and what role the OS plays in the system.

What could be a possible problem is when there are lots of agents in the system. Since each agent contains an Itinerary, and that two images are created when transferring an agent, the amount of overhead to store such information to persistent storage could be tremendous. Also, if lots of agents are created and deleted, synchronizing the name service provided by the Service Naming component across all Concordia Servers might prove difficult (synchronization of servers wasn't discussed in the papers).

Mole

Over all this system is very well done. However, there are few things that we would like to point out as we survey this system. We think that only providing support for Java is a flaw though it wasn't their goal to create a system anything besides Java. Second, messaging system uses session, which can be cumbersome and slow. Finally, strong migration should have been implemented incase a programmer willing accept the cost of using strong migration.

Tacoma

The current versions of Tacoma provide support for agents written in C, C++, ML, Perl, Python, Scheme, and Visual Basic. The TACOMA platform has also been ported to new operating system architectures, in particular Windows NT, Windows CE and the PalmOS. Several TACOMA applications are under construction [11]. One example is a wide-area network weather monitoring system accessible over the Internet. This distributed application is Stormcast. Whether agents can be useful in extensible file system architectures is under investigation.

With Tacoma, the only way to move an agent's state from one processor to another is by explicitly storing that state in one or more folders. Tacoma programmers must be cognizant of what state to capture and move; they must program these actions explicitly [11].

In higher level programming models where state is invisible to the programmer, automatic state capture becomes a necessity. The cost of moving an agent from one processor then cannot be predicted, and designing applications to meet performance goals becomes difficult.

The basic functionality offered by Tacoma has proven sufficient for a number of applications. However, two improvements to the basic functionality that are desirable to implement are 1) support for binary data and 2) to extend the semantics of *meet* to include communication with active agents [10]. Further work, which may increase performance

when data volume is sufficiently large, includes pre-copying and lazy copying schemes. Applying compression to briefcases could yield better performance in cases of low bandwidth. This compression can be made transparent to the application programmer. However, the system has to identify situations in which compression improves performance, rather than adding to the overhead. Alternative transport mechanisms such as UDP, SMTP, and HTTP could be incorporated into *meet*. The system could select between these when invoking *meet*, choosing the best suited.

Voyager

We are happy with this system; it has a lot more things that this system offers regarding the support with DCOM, RMI, and CORBA, which really stands out. However there is couple of things, which I didn't like in this system. First of all, it only uses java. Considering that it supports DCOM, and a lot of DCOM application use C++ code. Since voyager pure java, it doesn't support other languages, and that is something that limits the development languages. Second, messaging system in this system has holes in it. For example, what if one object keeps moving from machine to machine leaving a secretary behind, a message may take a very long time to get to the intended destination.

Conclusion

In this paper, we surveyed several mobile agent systems. While surveying these systems, we wondered why there were so many systems in the first place? Were all these systems in some sort of competition? Would not the efforts expended on developing new systems be better spent on improving an already existing system? Of course, the lack of certain required functionality cannot always be met within the most optimal manner with improvements in an existing system. Such a situation may warrant development of a new system.

Almost all the systems in this paper focused mainly on the environment provided by the system for agents, the mechanisms for agent mobility, agent communication, and language support. The use of mobile agents appears to offer certain advantages for client-server computing but as we've noted in the above systems, it also raises some difficult issues with respect to efficiency, flexibility and security. These issues have an effect on an agent's ability of mobility.

Many important issues such as how agents determine the available resources/services on a machine it transferred to (i.e., resource discovery), mobile agent system-to-OS interaction, the use of persistent storage (if any), and support for failure (i.e., fault-tolerance) were either briefly discussed or missing completely.

Mobile agents need more applications that take advantage of the characteristics of mobile agents since there is no single alternative to all of the functionality supported by a mobile agent framework. A potential application for mobile agents would involve the use of the Internet and the many uses of the Internet. Solutions to the security and virus problems in mobile agents could also result in new and successful methods of client-server interaction in network services.

The mobile agent approach continues to intrigue and definitely shows signs of offering important qualitative advantages for network services, but if there is some "killer application" that only mobile agents can provide then maybe they would be more ubiquitous and more readily used.

Bibliography

[1] David M. Chess, Colin G. Harrison, and Aaron Kershenbaum. "Mobile Agents: Are they a good idea?", IBM Research Report.

[2] Dag Johansen, Robert van Renesse and Fred B. Schneider, "Operating System Support for Mobile Agents", Department of Computer Science, Cornell University, USA, November 1994.

[3] David Kotz and Robert S. Gray. "Mobile Agents and the Future of the Internet", ACM Operating Systems Review, 33(3):7-13, August 1999.

[4] Robert S. Gray. "Agent Tcl: A transportable agent system", Proceedings of the CIKM Workshop on Intelligent Information Agents, Fourth International Conference on Information and Knowledge Management (CIKM 95), Baltimore, Maryland, December 1995.

[5] Robert S. Gray and David Kotz and George Cybenko and Daniela Rus. "D'Agents: Security in a multiple-language, mobile-agent system", In Giovanni Vigna, editor, Mobile Agents and Security, Lecture Notes in Computer Science, Springer-Verlag, 1998.

[6] David Wong, Noemi Paciorek, Tom Walsh, Joe DiCelie, Mike Young, and Bill Peet. "Concordia: An Infrastructure for Collaborating Mobile Agents", Mitsubishi Electric ITA, Horizon Systems Laboratory.

[7] "Mobile Agent Computing". A White Paper. Mitsubishi Electric ITA, Horizon Systems Laboratory, January 1998.

[8]"The Architecture of the Ara Platform for Mobile Agents", Holger Peine and Torsten Stolpmann. In proceedings of the first International Workshop on Mobile Agents, Berlin, Germany, 1997.

[9] "An Introduction to Mobile Agent Programming And The Ara System", Holger Peine. Technical report ZRI-Report 1/97, Department of Computer Science, University of Kaiserslautern, Germany, 1997.

[10] "TACOMA- Fundamental Abstractions supporting Agent Computing In A Distributed Environment" Nils Peter Sudmann, Department of Computer Science, University of Tromsø, Norway, November 1996.

[11] "What TACOMA Taught Us." Dag Johansen, Fred B. Schneider, and Robbert van Renesse. Also in, *"Mobility, Mobile Agents and Process Migration - An edited Collection"*, Dejan Milojicic, Frederick Douglis, and Richard Wheeler eds. Addison Wesley Publishing Company, 1998.

[12] "Overview of voyager", Graham Glass, CTO ObjectSpace

[13] "Voyager Core Package"

[14] "Mole- concepts of Mole Agents Systems", J. Baumann, F.Hohl, Dr. K. Rothermel, M. Straßer