Multimedia Services based on Mobile Agents

Anson Lee

Department of Computer Science and Enginneering
The Chinese University of Hong Kong
Email:cwlee@cse.cuhk.edu.hk

Abstract

This paper addresses the role and the advantages of mobile agents in the world of computing and discusses the use of mobile agents for the use in heterogeneous distributed multimedia services system. The main goal of the system is to combine the multimedia and the agent technologies: mobile agents can travel and execute on different nodes along the network and manipulate data of various media types such as audio and video.

1 Introduction

Distributed multimedia services systems tend to be heterogeneous. To make the heterogeneous environments work together, data is exchanged via networks. Standards have been designed to allow an exchange of information in a general manner. Standards are necessary in every layer of the information exchange model. They are in charge to enable to run programs on different hardware architectures. For communication between systems, several standardized architectures exist, such as ISO, TCP/IP, ATM, IPX, Banyan Vines, Appletalk and SNA.

At the application layer many combinations of underlying services are possible. With many standards in place it is possible for a requested layer stack to be not completely available to an applications. If the application cannot adapt itself to the configuration, it should be abandon. Therefore, two main problems for an application to run on a remote system can be observed:

1. Discovery of remote resources and services.

2. Configuration of the remote client or server.

High level languages such as scripting languages, available on every system can cover the heterogeneity of the machines, as they provide an unique execution platform. By adding the functionality to be mobile to a script, a script can operate on any machine. The script cannot only adapt to underlying services, but it can go to another location to create a common configuration between the local and the remote communicating system. Once the adaption is achieved, the communication can start. It might also be necessary to change a configuration for a communication with is already running. The scripts that reconfigure the systems will have to interact. Since mobile agents provide the flexibility to adapt themselves in an evolutionary environment, they can be used in many services and protocols defined to serve future applications.[1]

This paper discusses the Multimedia Transportable Agent System. The system consists of a multimedia agent platform and agent-based applications. It provides an agent transport protocol for agent migration and transfer of various media such as audio and video. The multimedia agent-based applications are built on top of the multimedia agent platform. The new applications offer services based on the agent paradigm. Agents can travel in a heterogeneous network, run on one machine, stop executing and update their state and migrate to a new machine. Agents can have different tasks such as retrieve or send multimedia documents. Finally, agents communicate with a set of defined instructions that constitute the inter-agent communication language.

2 Agent Environments

In this section, we introduce mobile agent and discuss its strength in the area of distributed information retrieval applications.

2.1 Mobile agents

A mobile agent is an executing program that can migrate during execution from machine to machine in a heterogeneous network. In other words, the agent can suspend its execution, migrate to another machine, and then resume execution on the new machine from the point at which it left off. On each machine, the agent interacts with stationary agents and other resources to accomplish its task.

Mobile agents have several advantages in distributed information retrieval applications. First, by migrating to the location of a needed resource, an agent can

interact with the resource without transmitting intermediate data across the network, conserving bandwidth and reducing latencies. Similarly, by migrating to the location of a user, an agent can respond to user actions rapidly. In either case, the agent can continue its interaction with the resource or user even if network connections go down temporarily. These features make mobile agents particularly attractive in mobile-computing applications, which often must deal with low-bandwidth, high-latency, and unreliable network links.

Second, mobile agents allow traditional clients and servers to offload work to each other, and to change who offloads to whom according to the capabilities and current loads of the client, server and network. Similarly, mobile agents allow an application to dynamically deploy its components to arbitrary network sites, and to re-deploy those components in response to changing network conditions.

Finally, most distributed applications fit naturally into the mobile agent model, since a mobile agent can migrate sequentially through a set of machines, send out a wave of child agents to visit machines in parallel, remain stationary and interact with resources remotely, or any combination of these three extremes. Complex, efficient and robust behaviors can be realized with surprisingly little code.

Although each of these advantages is a reasonable argument for mobile agents, none of them are unique to mobile agents, and, in fact, any specific applications can be implemented just as efficiently and robustly with more traditional techniques, such as message passing and remote invocation, process migration, remote evaluation, stored procedures, applets and servlets. Different applications require *different* traditional techniques, however, and many applications require a combination of techniques. In short, the true strength of mobile agents is not that they make new distributed applications possible, but rather that they allow a wide range of distributed applications to be implemented efficiently, robustly and easily within a single, general framework.

3 Survey of mobile agent systems

In this section, we examine nine representatives of mobile-agent system, and briefly discuss their similarities and differences.

3.1 Ara

Ara [2]supports agents written in Tcl and C/C++. The C/C++ agents are compiled into an efficient interpreted bytecode called MACE; this bytecode, rather than

the C/C++ code itself, is sent from the complete state of the agent, transfers the state to the target machine, and resumes agent execution at the destination. Ara also allows the agent to *checkpoint* its current internal state at any time during its execution. Unlike other multiple-language systems, the entire Ara system is multi-threaded; the agent server and both the Tcl and MACE interpreters run inside a single Unix process. Although this approach complicates the implementation, it has significant performance advantages, since there is little interpreter startup or communication overhead. When a new agent arrives, it simply begins execution in a new thread, and when one agent wants to communicate with another, it simply transfers the message structure to the target agent, rather than having to use inter-process communication. Nearly all Java-only systems are also multi-threaded, and have the same performance advantages.

3.2 D'Agents

D'Agents, which was also known as Agent Tcl[3], support agents written in Tcl, Java and Scheme, as well as stationary agents written in C and C++. Like Ara, D'Agents provides a `go` instruction (Tcl and Java only), and automatically captures and restores the complete state of a migrating agent. Unlike Ara, only the D'Agents server is multi-threaded; each agent is executed in a separate process, which simplifies the implementation considerably, but adds the overhead of inter-process communication. The D'Agent server uses public-key cryptography to authenticate the identity of an incoming agent's owner. Stationary resource-manager agents assign access rights to the agent based on this authentication and the administrator's preferences, and language-specific enforcement modules enforce the access rights, either preventing a violation from occurring or terminating the agent when a violation occurs. Each resource manager is associated with a specific resource such as the file system. The resource managers can be as complex as desired, but the default managers simply associate a list of access rights with each owner. Unlike Ara, most resource managers are not consulted when the agent arrives, but instead only when the agent attempts to access the corresponding resource or explicitly requests a specific access right. At that point, the resource manager forwards all relevant access rights to the enforcement module, and D'Agents behaves in the same way as Ara, enforcing the access rights with short wrapper functions around the resource access functions.

Current work on D'Agents falls into four broad categories: (1) scalability, (2) network-sensing and planning services, which allows an agent to choose the best migration strategy given the current network conditions; (3) market-based resource control, where agents are given a finite supply of currency from their owner's own finite supply and must spend the currency to access needed resources; and (4)

support for mobile- computing environments, where applications must deal with low-bandwidth, high-latency and unreliable network links.

D'Agents has been used in several information-retrieval applications, including the 3DBase, a system for retrieving three-dimensional drawings (CAD drawings) of mechanical parts based on their similarity to a query drawing.

3.3 Tacoma

Tacoma [4]supports agents written in C, C++, ML, Perl, Python, Scheme and Visual Basic. Unlike Ara and D'Agents, Tacoma does not provide automatic state-capture facilities. Instead, when an agent wants to migrate to a new machine, it creates a folder into which it packs its code and any desired state information. The folder is sent to the new machine, which starts up the necessary execution environment and then calls a known entry point within the agent's code to resume agent execution. Although this approach places the burden of state capture squarely onto the agent programmer, it also allows the rapid integration of new languages into the Tacoma system, since existing interpreters and virtual machines can be used without modification. Tacoma is used most notably in StormCast, which is a distributed weather monitoring system, and the Tacoma Image Server, which is a retrieval system for satellite images.

The public versions of Tacoma rely on the underlying operating system for security, but do provide hooks for adding a cryptographic authentication subsystem so that agents from untrusted parties can be rejected outright. In addition, the Tacoma group is exploring several interesting fault-tolerance and security mechanisms, such as using cooperating agents to search replicated databases in parallel and then securely vote on a final result, and using state machines to specify a machine's security policy and then directly using the state machines and software fault isolation to enforce the policy.

3.4 Java-Based systems

Aglets. Aglets [5]was one of the first Java-based systems. Like all commercial systems, including Concordia, Jumping Beans and Voyager, Aglets does not capture an agent's thread state during migration, since thread capture requires modifications to the standard Java virtual machine. In other words, thread capture means that the system could be used only with one specific virtual machine, significantly reducing market acceptance. Thus, rather than providing the `go` primitive of D'Agents and Ara, Aglets and the other commerical systems instead use variants of the Tacoma model, where agent execution is restarted from a known entry point

after each migration. In particular, Aglets uses an event-driven model. When an agent wants to migrate, it calls the `dispatch` method. The Aglets system calls the agent's `onDispatching` method, which performs application-specific cleanup, kills the agent's threads, serializes the agent's code and object state, and sends the code and object state to the new machine. On the new machine, the system calls the agent's `onArrival` method, which performs application-specific initialization, and then calls the agent's `run` method to restart agent execution.

Aglets includes a simple persistence facility, which allows an agent to write its code and object state to secondary storage and temporarily "deactivate" itself; proxies, which act as representatives for Aglets, and among other things, provide location transparency; lookup service for finding moving Aglets; and a range of message-passing facilities for inter-agent communication. The Aglet security model is similar to both the D'Agent and Ara security models, and to the security models for the other Java-based systems below. An Aglet has both an owner and a manufacturer. When the agent enters a context (i.e., a virtual place) on a particular machine, the context assigns a set of permissions to the agent based on its authenticated owner and manufacturer. These permissions are enforced with standard Java security mechanisms, such as a customized security manager.

Concordia. Concordia is a Java-based mobile agent system that has a strong focus on security and reliability. Like most other mobile-Java agent systems, they move the agent objects code and data, but not thread state, from one machine to another. Like many other systems, Concordia agents are bundled with an *itinerary* of places to visit, which can be adjusted by the agent. If the remote site is not currently reachable, agents, events and messages can be queued. Agents are carefully saved to a persistent store, before departing a site and after arriving at a new site, to avoid agent loss in the event of a machine crash. Agents are protected from tampering through encryption while they are in transmission or stored on disk; agent hosts are protected from malicious agents through cryptographic authentication of the agent's owner, and access control lists that guard each resource.

Jumping Beans. Jumping Beans is a Java-based framework for mobile agents. Computers wishing to host mobile agents run a Jumping Beans *agency*, which is associated with some Jumping Beans *domain*. Each domain has a central server, which authenticates the agencies joining the domain. Mobile agents move from agency to agency, and agents can send messages to other agents; both mechanisms are implemented by passing through the server. Thus the server becomes a central point for tracking, managing, and authenticating agents. It also becomes a central point of failure or a performance bottleneck, although they intend to develop scalable servers to run on parallel machines. Another approach to scalability is

to create many small domains, each with its own server. In the current version, agents cannot migrate between domains, but they intend to support that capability in future versions. Security and reliability appear to be important concerns of their system; public-key cryptography is used to authenticate agencies to the server, and vice versa; access-control lists are used to control an agent's access to resources, based on the permissions given to the agent's owning user.

Although they claim to move all agent code, data, and state, it is not clear from their documentation whether they actually move thread state, as in Agent Java. They require that the agent be a serializable object, so it seems likely that they implement the weaker form of mobility common to other Java-based agent systems.

3.5 Other systems

Messengers. The Messenger project uses mobile code to build flexible distributed systems, not specifically mobile-agent systems. In their system, computers run a minimal Messenger Operating System (MOS), which has just a few services. MOS can send and receive *messengers*, which are small packets of data and code written in their programming language `MO`. MOS can interpret `MO` programs, which may access one of their two bulletin-board services: the *global dictionary*, which allows data exchange between messengers, and the *service dictionary*, which is a searchable listing of messengers that offer services to other messengers. Ultimately, most services, including all distributed services, are offered by static and mobile messengers. In one case, they allow the messengers to carry native UNIX code, which is installed and executed on MOS; system calls are reflected back to the interpreted `MO` code, allowing fast execution of critical routines, while maintain-g the flexibility of mobile code.

Obliq. Obliq is an interpreted, lexically scoped, object-oriented language. An Obliq object is a collection of named fields that contain methods, aliases, and values. An object can be created at a remote site, cloned onto a remote site, or migrated with a combination of cloning and redirection. Implementing mobile agents on top of these mobile objects is straightforward. An agent consists of a user-defined procedure that takes a *briefcase* as its argument; the briefcase contains the Obliq objects that the procedure needs to perform its task. The agent migrates by sending its procedure and current briefcase to the target machine, which invokes the procedure to resume agent execution.

Visual Obliq builds on top of Obliq's migration capabilities. Visual Obliq is an interactive application builder that includes (1) a visual programming environment

for laying out graphical user interfaces, and (2) an agent server that allows Visual Obliq applications to migrate from machine to machine. When the application migrates, the state of its graphical interface is captured automatically, and recreated exactly on the new machine. Obliq does not address security issues. Visual Obliq does provide access control, namely, user-specified access checks associated with all "dangerous" Obliq commands, but does not have authentication or encryption mechanisms. Typically, therefore, the access checks will simply ask the user whether the agent should be allowed to perform the given action.

Telescript. Telescript is developed at General Magic, Inc., was the first commercial mobile-agent system, and the inspiration for many of the recent mobile-agent systems. In Telescript, each network site runs a server that maintains one or more virtual *places*. An incoming agent specifies which of the places it wants to enter. The place authenticates the identity of the agent's owner by examing the agent's cryptographic credentials, and then assigns a set of access rights or *permits* to the agent. One permit, for example, might specify a maximum agent lifetime, while another might specify a maximum amount of disk usage. An agent that attempts to violate its permits is terminated immediately. In addition to maintaining the places and enforcing the security constraints, the server continuously writes the internal state of executing agents to non-volatile store, so that the agents can be restored after a node failure.

A Telescript agent is written in an imperative, object-oriented language, which is similar to both Java and C++, and is compiled into bytecodes for a virtual machine that is part of each server. As in D'Agents and Ara, a Telescript agent migrates with the `go` instruction. A Telescript agent can communicate with other agents in two ways: (1) it can *meet* with an agent that is in the same place; two agents receive in a different place; the two agents then pass objects along the connection. Despite the fact that Telescript remains one of the most secure, fault-tolerand and efficient mobile-agent systems, it has been withdrawn from the market, largely because it was overwhelmed by the rapid spread of Java.

3.6 Similarities and differences

All mobile-agent systems have the same general architecture: a server on each machine accepts incoming agents, and for each agent, starts up an appropriate execution environment, loads the agent's state information into the environment, and resumes agent execution. Some systems, such as the Java-only systems above, have multi-threaded servers and run each agent in a *thread* of the server process itself; other systems have multi-process servers and run each agent in a separate

interpreter process; and the rest use some combination of these two extremes. D'Agents, for example, has a multi-threaded server to increase efficiency, but separate interpreter processes to simplify its implementation. Jumping Beans is of particular note since it uses a centralized server architecture (in which agents must pass through a central server on their way from one machine to another). rather than a peer-to-peer server architecture (in which agents move directly from one machine to another). Although this centralized server easily can become a performance bottleneck, it greatly simplifies, tracking, administration and other issues, perhaps increasing initial market acceptance.

Currently, for reasons of portability and security, nearly all mobile-agent systems either interpret their languages directly, or compile their languages into bytecodes and then interpret the bytecodes. Java, which is compiled into bytecodes for the Java virtual machine, is the most popular agent language, since (1) it is portable but reasonably efficient, (2) its existing security mechanisms allow the safe execution of untrusted code, and (3) it enjoys widespread market penetration. Java is used in all commercial systems and in several research systems. Due to the recognition that agents must execute at near-native speed to be competitive with traditional techniques in certain applications, however, several researchers are experimenting with "on-the-fly" compilation. The agent initially is compiled into bytecodes, but compiled into native code on each machine that it visits, either as soon as it arrives or while it is executing. The most recent Java virtual machines use on-the-fly compilation, and the Java-only mobile-agent systems, which are not tied to a specific virtual machine, can take immediate advantages of the execution speedup.

Mobile-agent systems generally provide one of two kinds of migration: (1) `go`, which captures an agent's object state, code, and control state, allowing it to continue execution from the exact point at which it left off; and (2) *entry point*, which captures only the agent's object state and code, and then calls a known entry point inside its code to restart the agent on the new machine. The `go` model is more convenient for the end programmer, but more work for the system developer since routines to capture control state must be added to existing interpreters. All commerical Java-based systems use entry-point migration, since market concerns demand that these systems run on top of unmodified Java virtual machines. Research systems use both migration techniques.

Finally, existing mobile-agent systems focus on protecting an *individual* machine against malicious agents. Aside from encrypting an agent in transit and allowing an agent to authenticate the destination machine before migrating, most existing systems do not provide any protection for the agent or for a group of machines

that is not under single administrative control.

Other differences exist among the mobile-agent systems, such as the granularity of their communication mechanisms, whether they are built on top of or can interact with CORBA, and whether they conform to the emerging mobile-agent standards. Despite these differences, however, all of the systems discussed above (with the exception of Messengers, which is a lighter-weight mobile-agent system) are intended for the same applications, such as workflow, network management, and automated software installation. All of the systems are suitable for distributed information retrieval, and the decision of which one to use must be based on the desired implementation language, the needed level of security, and the needed performance.

4 A Distributed Multimedia Service System

Multimedia services cannot be modeled as the classic data communication services. They have different characteristics, as their basic function is to realize a communication between persons or from person to a machine, but not a machine to machine communication. In this section, we will study the architecture of our distributed multimedia service system and the current status of the prototype we are developing.

4.1 The agent structure

The structure of the mobile agent can be divided into three main parts: The transport header, the agent header and the agent script. Every application must apply this structure to its mobile agents.

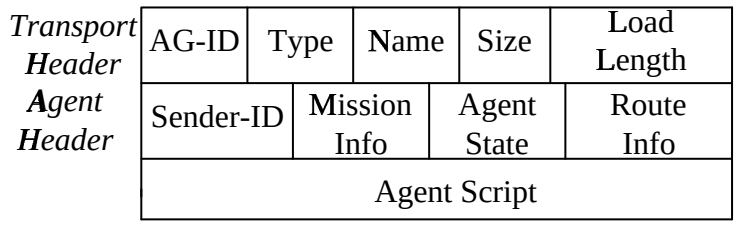

Figure 1: The agent structure.

The transport header is added to every file (not only agent files) sent in the network to a new destination and consists of five elements:

- The agent identification (AG-ID) is unique for every agent in the network. This number is used to name the directory where the agent and the load are stored. This AG-ID is used for reference when an agent or any file is lost in the network.
- The Type determines the type of the file. It is used to distinguish between agent files and the different media files in the agent load.
- The Name of the file.
- The Size of the file.
- The load length indicates the total number of files in the load that the agent owns and transports.

The agent header contains four elements:

- Sender Identification such as the user name, company, e-mail address.
- Mission Information describes the mission, and parameters not to exceed such as the maximum number of sites to visit, the maximum bytes to carry in the load etc.
- The agent state such as the size of the load carried, the number of sites already visited, the type of information found.
- Route Info is a list of service site addresses that will guide the agent during its mission.

4.2 Multimedia Server Infrastructure

Integrating mobile agents into a Multimedia server means to enable the servers to start, receive and execute mobile agents. Additionally, mobile agents may access the multimedia data of the server, may want to communicate with a user, and have to be managed. Starting, executing accessing multimedia data, and management of agents is done by a special extension module: the Multimedia Agent Platform(MAP). In this way, the code of multimedia server does not need to be changed.

We choose C and Agent Tcl as the implementation language for our system and for agent-programming. Since Agent Tcl can be embedded in C, this allows us to customize and start a mobile agent easily.

Figure 2 shows the overall architecture of the Multimedia Server Infrastructure which consists of an user interface, our multimedia server and the server's MAP.

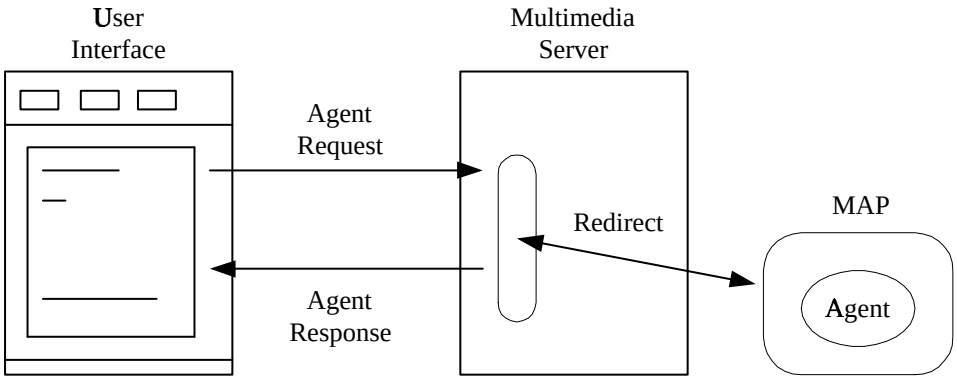

Figure 2: Multimedia Server Infrastructure.

When the server receives an agent related request, the server hands it to its MAP to fulfill the request. There is no need for the server to understand any MAP related request, it is sufficient that it identifies the request as a MAP request to forward it to its MAP. In this way, any changes in MAP requests do not affect the server implementation.[6] [7]

Generally speaking, our scenario is divided into two phases: the agent startup and configuration phase, and the agent service phase. In the first phase, a user wants to access an agent which is located at a multimedia server. The user uses the graphical user interface to connect to the server and requests the start of the mobile agent. Since all agent related actions are executed by the MAP, the server redirects the user's request to it. The MAP inits and starts the mobile agent. After that, the second phase begins: depending on the configuration, the service is provided by executing the mobile agent, such as migrating to other multimedia servers to access their multimedia data. After completing its task, the agent may return to the server by which it was started and notify the user about the result of its work.

4.3 Multimedia Agent Platform Architecture

The architecture of the MAP is showed in fig. 3. It has two main components: The migration facilitator and the Agent Execution Environment.

The migration facilitator is responsible for the transport of the agent and its load. The agent load can include many files of different media types such as audio or video. The migration facilitator is implemented through a Agent Tcl daemon.[3] It listens to a well known port for incoming agents. For departing agents, the migration facilitator in the sender machine is connected to the migration facilitator in the destination machine and transfers the agent file and all the files in the load

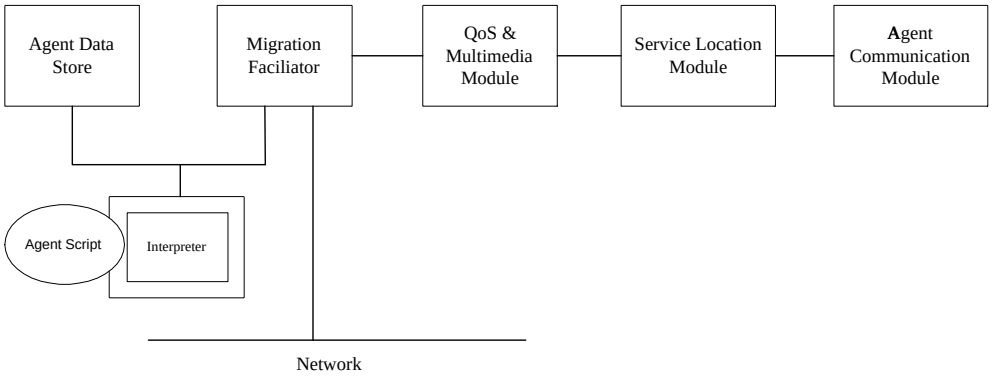

Figure 3: Architecture of the multimedia agent platform.

to the new destination. For every incoming agent, the migration facilitator reads the agent header in the agent structure, creates a directory in the agent data store and stores the agent and load files. If the migration of the agent is successful, the migration facilitator activates the Agent Execution Environment and waits for new incoming agents.

The Agent Execution Environment recognizes the structure of the agent and extracts information necessary to run the agent: the mission description, the sender identification, the agent state and the script. Agent scripts are built using Agent Tcl as interpreted languages.[3] Interpreted languages offer many advantages. The most important advantage is security: It is much easier to check malicious script as it is being interpreted instruction by instruction than a big chuck of a binary code. The second advantage is portability. Interpreted programs can run on any platforms.

The Quality of Service (Qos) and Multimedia Module allows for negotiation of bit rate. The transportation of multimedia information faces mainly one problem. A media stream is continuous and of high bandwidth need, which results in huge amounts of data to transport.

We propose to use a mobile agent to negotiation with the client side to solve that problem. The solution is done in the Qos module. We can get the feedback from the receiver which include the available buffer space and the network bandwidth, the server is able to modify or change the coding scheme to lower or raise the quality and the bandwidth needs for the stream. As a result, the application allows to adapt to the momentary network load from end to end.

To establish a multimedia communication several phases are necessary. First,

the remote site has to be configured by the mobile agent. Local resources have to be reserved. This can be done by the negotiation between two Qos modules in the communication parties.

Second, the network connection between the server and the client has to be setup.

Third, the communication Qos during the transmission of the multimedia stream can be modified.

Fourth, the sessions ends. The local resources have to be free and the network connection have to be closed.

4.4 Agent Startup

As discussed in the earlier section, the structure of a mobile agent can be divided into three parts. In order to configure the mobile agent by the user at the startup stage, we create several graphical user interface for the agents. They are written by Tk scripts which are embeded in the agent script inside the agent structure. The agent may choose its graphical user interface depending on a startup parameter. This can create different views to the same functionality.

Figure 4 shows the agent startup scenario. When a user requests the start of an agent by a get request to the multimedia server, the server hands this request to its MAP that identifies and loads the agent. After loading and starting, the agent sends its user interface to the client side [8]and let the user to configure the agent. After the user is done, the configuration data is sent back to server and stores in the agent.

The startup scenario can be achieved easily using Agent Tcl [3]as the agent interpreter. The load and start of an agent can be done by using the `agent_begin` command. The `agent_begin` command registers the agent with the agent server on the specified machine (or on the local machine if no machine is specified) and returns the agent's new identifier within the agent namespace. In Agent Tcl 2.0, this identifier consists of the symbolic name of the server, the IP address of the server, a symbolic name that the agent chooses for itself, and a unique integer that the server assigns to the agent. So if an agent issues the command `agent_begin pc89130`, for example, the command might return the four-element Tcl list `pc89130.cs.cuhk.edu.hk 137.189.89.130 {} 15`. The `137.189.89.130` is the IP address of `pc89130`. The empty curly brackets indicate that the agent initially has no symbolic name; a symbolic name can be chosen at a later time with

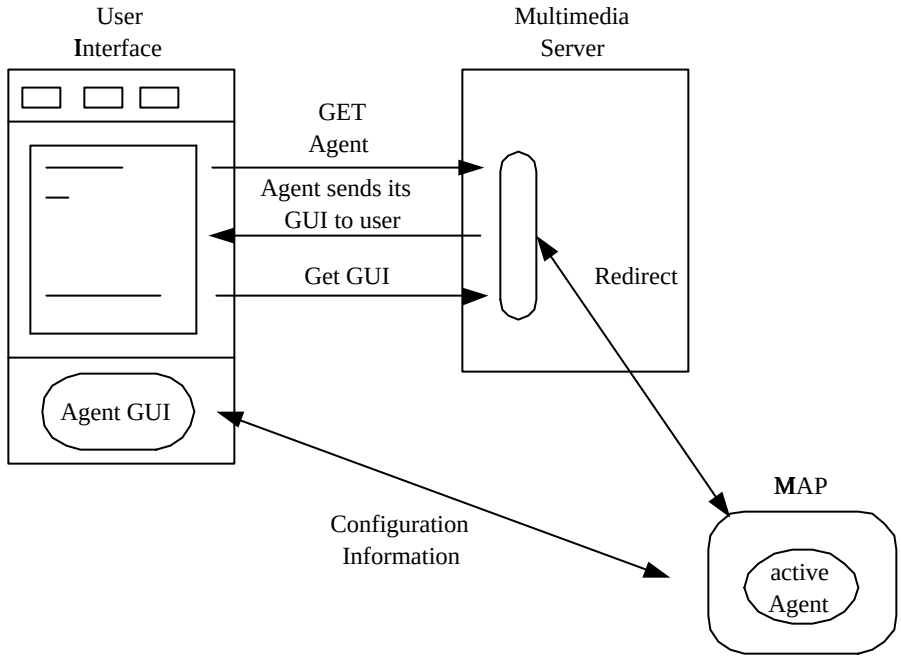

Figure 4: Agent Startup Scenario.

the `agent_name` command. The `15` is the integer id that the migration facilitator on `pc89130` has assigned to the new agent; this integer is unique among all agents executing on `pc89130` but not among all agents everywhere. Once the agent has issued the `agent_begin` command, it can use the other agent commands.

4.5 Agent Migration

The migration scenario is managed by the migration facilitator in the MAP. Again, the use of Agent Tcl makes the implementation easy for us. There is a category of commands which allow an agent to migrate from machine to machine and to create child agents.

An agent can call the `agent_jump` command when it wants to migrate to a new machine. This command captures the internal state of the agent and sends the state to the next migration facilitator on the specified machine. The migration facilitator in the next machine restores the state and the agent continues execution immediately after the `agent_jump`. Certain components of the state, such as open files and child processes, are tied to a specific machine and are not transferred to the new machine.

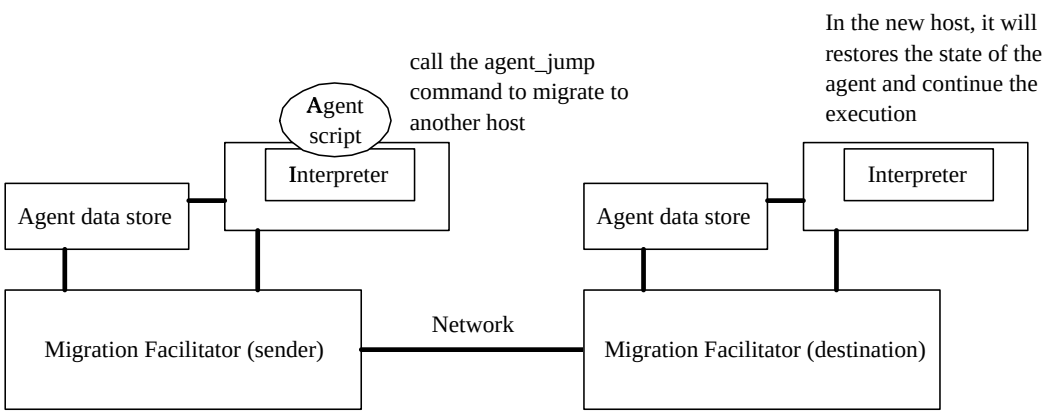

Figure 5: Agent Migration Scenario.

4.6 Agent Communication and Service Location Protocol

This is an essential part for the mobility of agents. As the mobile agents move around, it is better to point them directly to the destination they are looking for, than letting them search without any guidance. An agent environment should provide this facility to delegate the resource discovery problem.

The service location protocol eliminates the need for a mobile agent to travel to a destination which doesn't have the requested resources. Rather, the mobile agent supplies a set of attributes which describe the service to a service agent. They can get information about destinations, such as the next hop address, based on higher level service requests.

A mobile agent, that wants or needs to be known to other agents, registers itself with the service agent. Other agents that are looking for it can therefore find it and exchange messages. It is necessary to have one service agent on every MAP, to be able to register and unregister the agents which are currently on the machine.

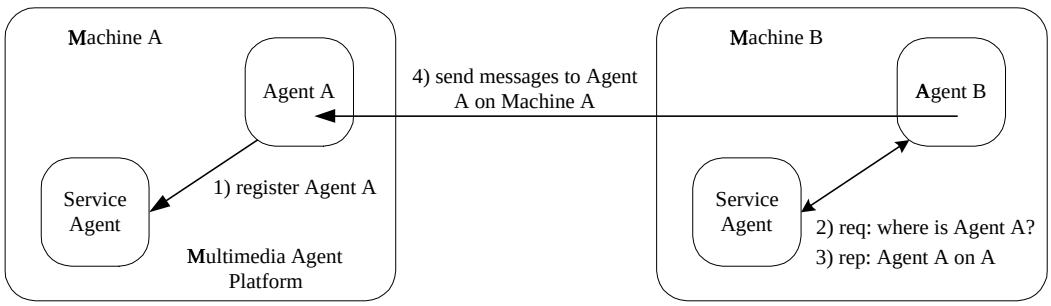

Figure 6: Agent Communication Scenario.

4.7 Agent-Based Applications

We are implementing an agent-based multimedia application built on top of the MAP described above. The application supports mobile agents with the following missions:

- Search mp3 audio files and mpeg video files by matching keywords.
- Stream and Play audio files and video files on a remote location.

The application is developed under Redhat Linux 6.2 with no mobile agent technology applied at this moment. But the streaming of mp3 audio file and MPEG 1 video files is tested successfully over a LAN. The implementation details of the application will be discussed in the following section.

4.7.1 Audio Delivery

The audio delivery application is implemented under Redhat 6.2. We use MPEG 1 layer III (mp3) compression techniques to compress the audio file. The mp3 files are stored in the same machine where the audio server is running. The audio client is a software mp3 decoder. The bit rate of the mp3 files is 128kbps. [9][10]

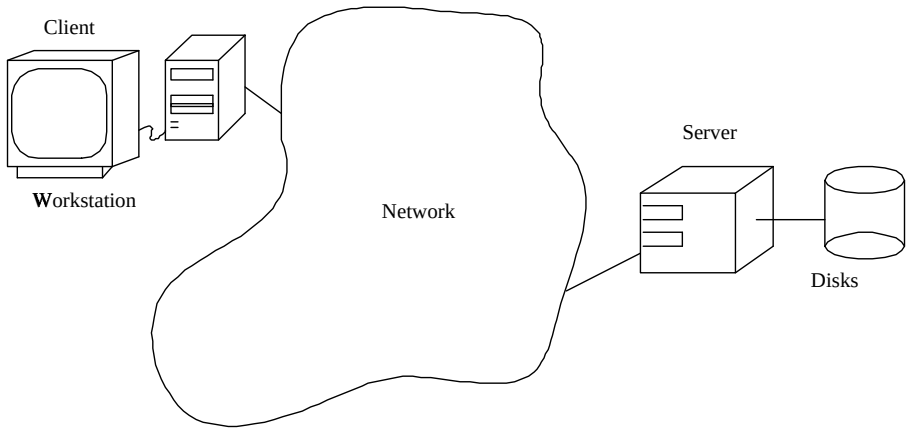

Figure 7: Audio Delivery System.

Compression layers. MPEG/audio offers a choice of three independent layers of compression. This provide a wide range of trade-offs between codec complexity and compressed audio quality.

Layer I, the simplest, best suits bit rates above 128Kbps per channel. For example, Philips' Digital Compact Cassette (DCC) uses Layer I compression at

192Kbps per channel.

Layer II has an intermediate complexity and targets bit rates around 128Kbps per channel. Possible applications for this layer include the coding of audio for digital audio broadcasting and Video CD.

Layer III is the most complex but offers the best audio quality, particularly for bit rates around 64Kbps per channel. This layer suits audio transmission over ISDN.

All three layers are simple enough to allow single-chip, real-time decoder implementations.

Audio decoder. Figure 6 shows the block diagram of the MPEG audio decoder in our application. The decoder deciphers the encoded bitstream, restores the quantized sub-band values, and reconstructs the audio signal from the sub-band values.

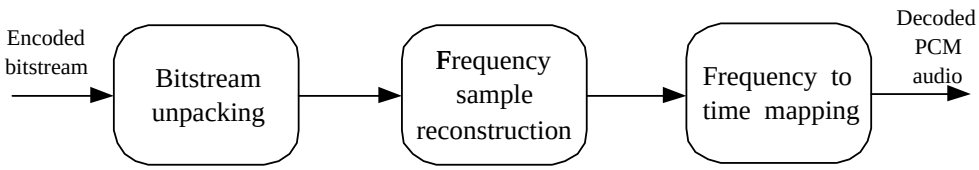

Figure 8: MPEG audio decoder.

Transmission Plan. In order to meet the bit rate requirement of the mp3 compression techniques during the streaming of mp3 files, we use the following transmission plan. First of all, we need to calculate the sleep interval between each sending attempt. The result should be 512bytes/128kbps, which is equal to 32000μsec.

Suppose we transmit the first packet (512bytes) of audio data at time t=0, the next transmit attempts should start at t=32000μsec. In the calculation of the sleep interval, we do not take the transmission overhead and the context switching time into account. Hence, it is a must to compute a new sleep interval after the first attempt with a view to compensate the latency in time.

Figure 8 shows the above scenario. When we transmit the second packet of audio data, t must be less than 32000μsec. Hence the next sleep interval must be smaller than the first one. In this case, the new sleep interval should be equal to

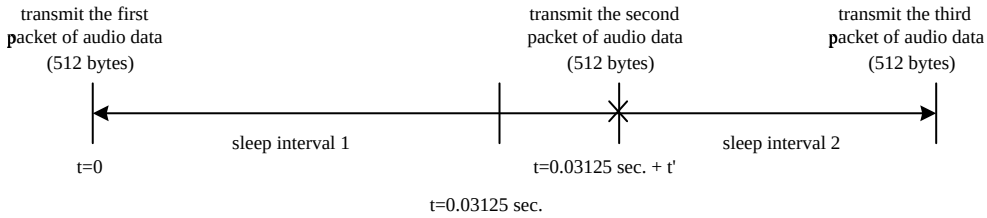

Figure 9: Calculation of sleep interval.

$\frac{1}{512/t}$.

Below is the pseudo code of this simple transmission plan.

```
sleep interval t=0.03125sec;
record the start time;
transmit audio packet;
record the stop time;
sleep for t sec;

while(server buffer not empty){
      compute next sleep interval t';
      record the start time;
      transmit audio packet;
      record the stop time;
      sleep for t' sec;
}
```

Using this transmission plan, we can approximate the bit rate to 128kbps which looks like the following figure.

Buffer Management in Client Side. Statistical results show that the bit rate of some audio files are greater than 128kbps and some packets may arrived at the client side in burst. Hence, we must implement a buffer in front of the audio decoder. We use a leaky bucket model in the buffer. The use of the leaky bucket model is to provide a constant output bit rate to the audio decoder.

Conceptually, the leaky bucket model is similar to a single-server queuing system with constant service time. If a packet arrives at the queue when it is full, the packet is discarded. This mechanism turns an uneven flow of audio packets from the network into an even flow of audio packets to the decoder.

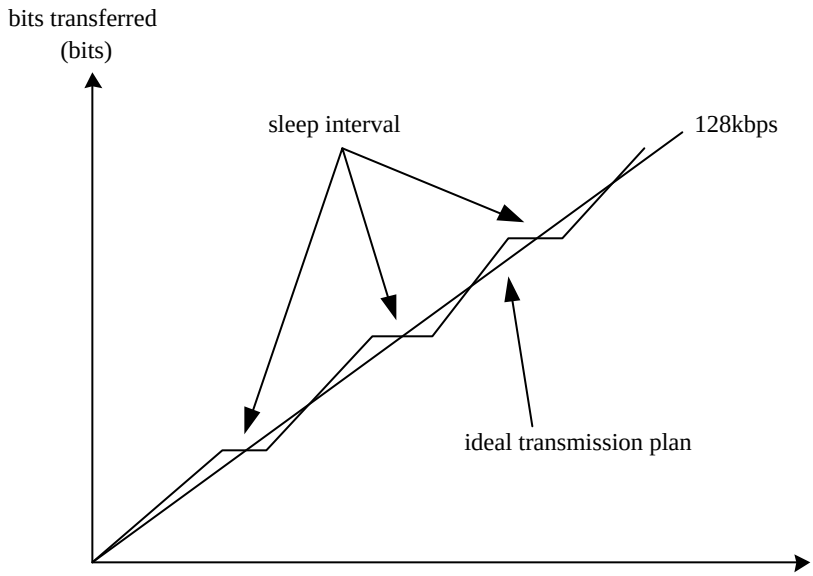

Figure 10: Transmission plan.

Implementing the leaky bucket algorithm is easy. The leaky bucket consists of a finite queue. When a packet arrives, if there is room on the queue it is appended to the queue; otherwise, it is discarded. At every clock tick, one packet is transmitted to the decoder.

As an example of a leaky bucket, imagine that a host machine receives audio packet at 130kbps. To reduce the average rate to 128kbps, we could use a leaky bucket with ρ = 128kbps and a capacity of 200kb. This means that the bursts of up to 200kb can be handled without data loss, and such bursts are spread out over 0.64 sec, no matter how fast they come in.

4.7.2 Video Delivery

The video delivery application is implemented under Redhat 6.2. We use MPEG 1 video compression standard to compress the video file. The video files are stored in the same machine where the video server is running. The video client is a software MPEG 1 decoder. The bit rate of the MPEG 1 video files is 1.5Mbps.

MPEG 1 Video Overview. The basic idea behind MPEG video compression is to remove spatial redundancy within a video frame and temporal redundancy between video frames. As in JPEG, the standard for still image compression, DCT-based compression is used to reduce spatial reddudancy. Motion compensation is

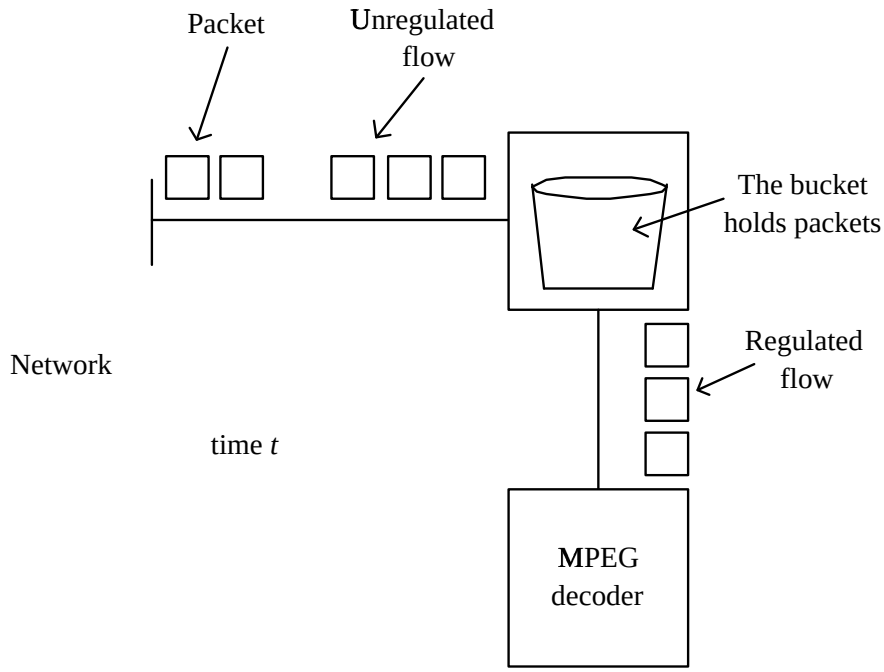

Figure 11: Leaky Bucket model in client side.

used to exploit temporal redundancy. The images in a video stream usually do not change much within small time intervals. The idea of motion compensation is to encode a video frame based on other video frames temporally close to it.

Video Decoder. The current implementation of the video client is make use of MpegTV. It is one of the world's leading companies for MPEG video and audio software playback across various architectures and operating systems.

The MPEG bitstream consists of a sequence of packs that are in turn subdivided into packets, as sketched in Figure. Each pack consists of a unique 32-bit byte-aligned pack start code and header, followed by one or more packets of data. Each packet consists of a packet start code and header, followed by packet data. the system decoder parses this bitstream and feeds the separated video and audio data to the appropriate decoders along with timing information.

Transmission Plan. We are implemented a simple transmission plan in video delivery. The concept is the same as audio delivery. The bit rate requirement of MPEG 1 video is 1.5Mbps. Hence, some parameters in calculting the sleep interval should be changed.

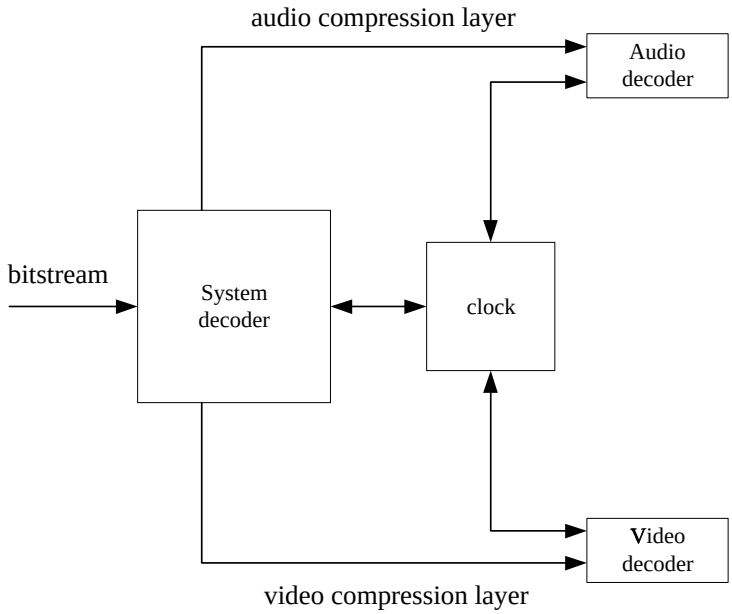

Figure 12: MPEG system structure.

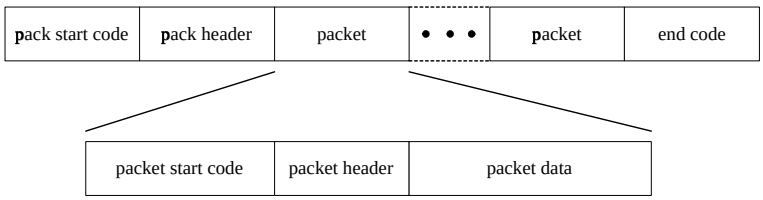

Figure 13: System layer pack and packet structure.

Ideally, the time interval between each sending attempts should be equal to 512bytes/1.5Mbps = 2730μsec.

Again, when we transmit the second packet of video data, t must be less than 2730μsec. Hence the next sleep interval must be smaller than the first one. In this case, the new sleep interval should be equal to $\frac{1}{512/t}$.

Using this simple transmission plan, we can approximate the transmission curve to the ideal one just like the audio case.

5 Conclusion

In this paper, we have discussed different types of Agent Environment and compared their similarities and differences. Besides, we have discussed the role and the advantages of mobile agents in a heterogeneous distributed multimedia services system. The main goal of the system is to combine the multimedia and the agent technologies. The multimedia server architecture with mobile agent services has been proposed. Two applications including audio delivery and video delivery have been developed based on the multimedia agent platform.

6 Future Work

The implementation of the whole system is still in progress. However, the audio and video delivery applications are completed. Future work can include protecting mobile agent using some encryption algorithms.

References

[1] B. Brewington, R. Gray, K. Moizumi, david Kotz, G. Cybenko, and D. Rus, "Mobile agents in distributed information retrieval," *Spriner Verlag*, pp. 355–395, 1999.

[2] P. H., "The ara projekt," http://www.uni-kl.de/AG-Nehmer/Ara/ara.html.

[3] G. R.S., "Agent tcl: A flexible and secure mobile-agent system,," in *Proc. of the Fourth Annual Tcl/Tk Workshop, Monteray CA*, 1996. http://www.cs.dartmouth.edu/agent/papers.html.

[4] J. D., "An introduction to the tacoma distributed system," technical report, Univ. of Tromso, Dept. of CS, June 1995.

[5] L. D. and C. D.T., "Ibm aglets workbench - programming mobile agents in java," *White Paper, IBM Corporation, Japan*, 1996.

[6] S. Funfrocken, "How to integrate mobile agents into web servers," in *Proc. IEEE 1997 Workshop on Collaborative Agents in Distributed Web Applications*, 1997.

[7] B.-S. A., "Jigsaw java http server," http://www.w3.org/pub/WWW/Jigsaw.

[8] J. K. Ousterhout, *Tcl and the Tk Toolkit*. Massachusetts, USA: Addison-Wesley Professional Computing Series, 1994.

[9] D. Pan, "A tutorial on mpeg/audio compression," *IEEE Multimedia*, pp. 60–74, 1995.

[10] P. Noll, "Mpeg digital audio coding," *IEEE Signal Processing Magazine*, pp. 59–81, 1997.