5-19-1995

# Ph.D. Thesis Proprosal: Transportable Agents

Robert S. Gray
*Dartmouth College*

# Ph.D. Thesis Proposal: Transportable Agents

Robert S. Gray
Department of Computer Science
Dartmouth College
6211 Sudikoff Laboratory
Hanover, New Hampshire 03755

19 May 1995

**Abstract**

One of the paradigms that has been suggested for allowing efficient access to remote resources is *transportable agents*. A transportable agent is a named program that can migrate from machine to machine in a heterogeneous network. The program chooses when and where to migrate. It can suspend its execution at an arbitrary point, transport to another machine and resume execution on the new machine. Transportable agents have several advantages over the traditional client/server model. Transportable agents consume less network bandwidth and do not require a connection between communicating machines – this is attractive in all networks and particularly attractive in wireless networks. Transportable agents are a convenient paradigm for distributed computing since they hide the communication channels but not the location of the computation. Transportable agents allow clients and servers to program each other. However transportable agents pose numerous challenges such as security, privacy and efficiency. Existing transportable agent systems do not meet all of these challenges. In addition there has been no formal characterization of the performance of transportable agents. This thesis addresses these weakness. The thesis has two parts – (1) formally characterize the performance of transportable agents through mathematical analysis and network simulation and (2) implement a complete transportable agent system.

Thesis committee

Dr. George Cybenko, Thayer School of Engineering, Dartmouth College
Dr. David Kotz, Department of Computer Science, Dartmouth College
Dr. Daniela Rus, Department of Computer Science, Dartmouth College
Dr. Robert Sproull, Sun Microsystems

# Contents

# List of Figures



# List of Tables

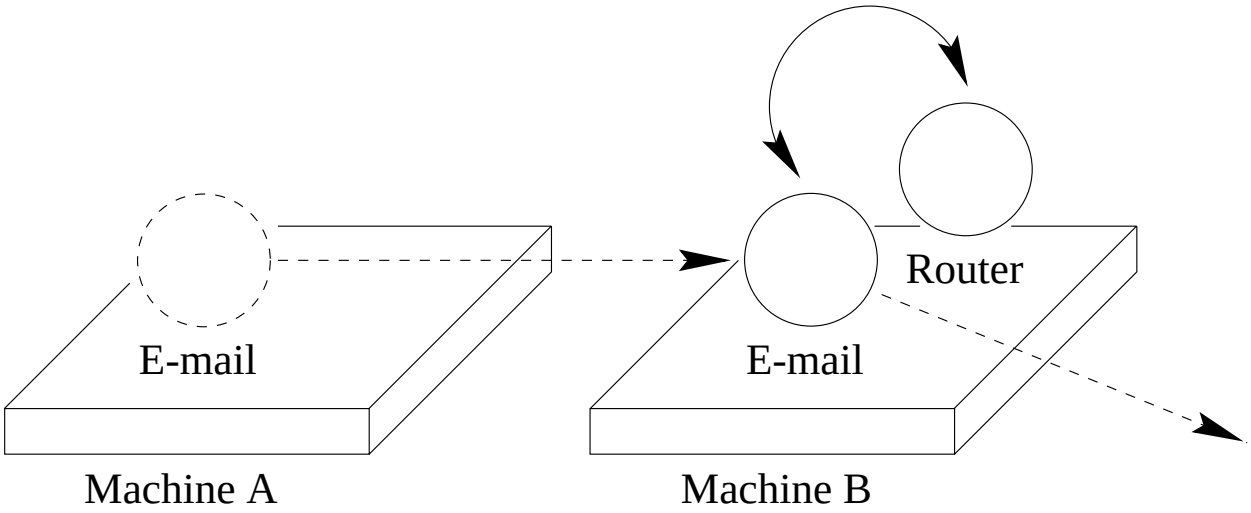



Figure 1: The transportable agent abstraction – an agent *jumps* from machine to machine and interacts with resources on each machine. In this case an active E-mail message has jumped to interact with a router and will jump again to interact with the recipient's mailbox. This figure was adapted from [Whi94].

# 1 Introduction

One of the paradigms that has been suggested for allowing efficient access to remote resources is *transportable agents*. A transportable agent is a named program that can migrate from machine to machine in a heterogeneous network. The program chooses when and where to migrate. It can suspend its execution at an arbitrary point, transport to another machine and resume execution on the new machine. Although the idea of a program that can move from machine to machine *under its own control* is not new [WVF89], it is only in the last two years that production-quality systems have been implemented. The most notable is General Magic's Telescript$^{TM}$ [Whi94] which is being used in AT&T's new PersonaLink$^{TM}$ network [Rei94]. The recent interest in transportable agents has been fueled by the growing inadequacy of the traditional client/server model for distributed applications.

In the traditional client/server model the server provides a fixed set of operations. All operations that are not provided in this fixed set must be performed at the client. If the server does not provide an operation that matches the client task exactly, either the client must make a series of remote calls to the server or a programmer must add a new operation to the server. The first option brings intermediate data across the network on each call. This represents significant wasted bandwidth if the data is not useful beyond the end of the client task. The second option is an intractable programming task as the number of clients grows. In addition it discourages modern software engineering. The server becomes a collection of complex, specialized routines rather than a collection of simple primitives. Transportable agents avoid this wasted bandwidth and allow efficient execution even when the server does not provide specialized operations. This is because a transportable agent migrates *to the server* where it can perform any desired processing before returning the final result to the client. Agents that do more work avoid more intermediate messages and conserve more network bandwidth. The performance advantage is greatest in low bandwidth and high latency networks [Whi94].

Modern client/server models such as remote evaluation and SUPRA-RPC address the same problem by sending subprograms to the server for evaluation. However these subprograms are anonymous entities. There is no convenient way for two subprograms to communicate with each other which makes it difficult to share partial results. At worst a subprogram must be divided into multiple subprograms. At best the partial results must be transmitted twice (once as they are sent to the home machine and once as the home machine distributes the partial results to the appropriate remote subprograms). Most transportable agent systems support direct communication between agents and avoid this extra communication overhead when

exchanging partial results.

Transportable agents do not require a connection between the local and remote sites and do not require state information at both sites. This makes transportable agents more fault-tolerant [WVF89] and in combination with their low use of network bandwidth, makes them ideally suited to mobile computing [Whi94]. Mobile computing is characterized by low bandwidth, high latency and periods of disconnection from the network.

Transportable agents are a convenient paradigm for distributed computing. First, the communication channels are hidden from the programmer but the location of the computation is not [JvRS95]. The agent specifies when and where to migrate and the system handles the transmission. This makes transportable agents easier to use than low-level facilities in which the programmer must explicitly handle communication *but* more flexible and powerful than schemes such as process migration in which the system decides when to move a program based on a small set of fixed criteria. Second, transportable agents allow the implicit transfer of information. A migrating agent carries all of its internal state with it which eliminates the need for separate communication steps. Third, many tasks – especially network management, information retrieval and workflow – fit naturally into the *jump-do-jump* model of transportable agents. The agent migrates to a machine, performs a task, migrates to a new machine, performs a task that might be dependent on the outcome of the first task and so on. Finally, transportable agents allow a distributed application to be written as a single program.

Transportable agents are useful even when viewed as an extension of the traditional client/server model. Clients and servers can program each other which greatly extends the functionality that application and server developers can provide to their customers. In addition an application can *dynamically* distribute its server components when it starts execution [JvRS95].

Applications that have been suggested for transportable agents include distributed information retrieval, network management, active e-mail, active documents, controlling remote devices, workflow and electronic shopping [Ous95, Whi94]. Any application in which it makes sense to move the program to the remote data or resource is a candidate for the transportable agent paradigm. The limiting factors are technical. For example there must be sufficient security precautions so that malicious agents can not damage a resource. The main weakness of existing systems is that they do not meet all of the technical challenges. They artificially limit the set of potential applications by not providing sufficient security, flexibility and efficiency. The second weakness of existing work is that there has been no formal characterization of the relationship between network conditions and agent performance. This makes it difficult to select the appropriate implementation when developing distributed applications and – just as importantly – makes it difficult for a transportable agent to select an appropriate communication strategy in the face of changing network conditions.

This thesis seeks to address these two weaknesses. First we seek to develop a formal model for transportable agents and use this model to explore agent performance under different network, data and application characteristics. This exploration will consists of mathematical analysis and extensive simulation since the model is likely to be complex enough to make most mathematical analysis intractable. Second we seek to implement a complete transportable agent system that meets all of the technical challenges. Particular areas of concern are security, privacy, efficiency, a flexible development environment and low-level support for high-level agent tasks such as negotiation, coordination and planning. Partial solutions are scattered throughout the distributed computing, operating system, programming language, artificial intelligence and transportable agent literature. These solutions need to be identified, extended and integrated into a coherent whole. The implementation will be evaluated in light of the formal model. The evaluation consists of instrumenting the system and representative agent applications in order to confirm or reject and rework the model. An accurate model should be made available to agents so that they can select the appropriate communication strategy for current network conditions.

Section 2 discusses existing transportable agent systems and related work. Section 3 explains the proposed thesis work. Section 4 provides an overview of the existing prototype. A schedule and documentation for the prototype are attached as appendices.

# 2 Related work

The popular definition of an agent is an intelligent software servant that either (1) relieves the user of routine, burdensome tasks such as appointment scheduling and e-mail disposition or (2) filters the overwhelming volume of electronic information so that the user sees only the information that is relevant to her current interests and needs [Haf95, Rog95]. This definition – due to its broadness and its ability to capture the imagination – has made "agent" a buzzword within both the academic and commercial worlds. Applications are often described as agent-based solely to draw attention or increase sales. For example *No Hands Software* describes its Magnet$^{TM}$ program as the "first intelligent agent for the Macintosh" even though it is essentially a file-finder [Fon93]. This inappropriate use of the term makes it difficult to separate hype from actual research. However there appear to be five legitimate contexts in which the term "agent" is used – *artificial intelligence*, *personal assistants*, *distributed information retrieval*, *software engineering and interoperation* and *transportable agents*. These five contexts are far from disjoint.

First we present brief examples of personal assistants and the agent-based approach to distributed information retrieval. This underscores the difference between the various kinds of agents and illustrates potential applications for transportable agents. Then we discuss the agent-based approach to software interoperation in detail since the ideas developed in this area will be critical in allowing effective communication between transportable agents. Finally we describe the precursors to transportable agents and existing transportable agent systems.

## 2.1 Artificial intelligence

Here an *agent* is an entity that perceives its environment with sensors and acts on its environment with effectors [RN95]. Such an agent can be either hardware with physical sensors and effectors or software with simulated sensors and effectors. This definition of an agent is used when attempting to provide a unified framework for artificial intelligence, when discussing software artifacts from a robotics viewpoint and of course when discussing physical robots. This definition is not considered further except to note that it subsumes the definitions below. The term *agent* is also used to describe certain kinds of artificial intelligence programs such as programs that automatically negotiate with each other in order to obtain a desired service. Automated negotiation will play an important role in a commercial transportable agent system.

## 2.2 Personal assistants

Here an *agent* is a program that relieves the user of a routine, burdensome task such as appointment scheduling or e-mail disposition. These agents are distinguished from traditional utilities by (1) their use of machine learning so that they can adapt to user habits and preferences [Mae94] or (2) their use of automated reasoning so that they can make complex inferences about the work environment [Rie94].

Maes presents a series of four agents that start with a minimum amount of domain knowledge and learn how to perform the task by observing and interacting with the user and other agents [Mae94]. This approach addresses the difficult problems of *competence* and *trust*. The agent is competent to perform the task *for a specific user* because it has learned to mimic user decisions. The user trusts the agent because it gradually takes over the performance of the task and makes the same decisions that the user would. The four agents are an electronic mail handler, a meeting scheduler, an electronic news filter and a reviewer that recommends books, music and other entertainment. The electronic mail handler uses memory-based reasoning to filter mail. It remembers every situation-action pair that occurs when the user filters her mail manually – e.g. she deleted a certain message. When a new situation occurs, the agent predicts what action should be taken by comparing the new situation with the nearest memorized neighbors. Depending on its confidence in the prediction, the agent will perform the action, suggest the action to the user or let the user handle the situation herself. In addition to learning from example, the agent accepts directives from the user and solicits suggestions from other mail agents. The other three agents share the same basic design.

Riecken has developed a more complex system called *M* that is designed to automatically group the doc-

uments that are presented during the course of a virtual multimedia conference [Rie94]. The heart of the system is five inference engines – functional, structural, causal, spatial and temporal – that attempt to infer relationships among documents based on the actions that the users apply to the documents. For example if two documents are placed close together within the virtual conference room, the spatial engine might suggest that the documents deal with the same subject. The engines develop theories on a dynamic set of blackboards. Each blackboard contains one potential theory that explains the relationships among the documents. The engines post their conclusions to the blackboards and use the conclusions of the other engines to continue the reasoning process and determine the correctness of the competing interpretations. Eventually one interpretation emerges as the most likely.

A transportable agent system should be able to support these applications in a distributed setting. At a minimum this means that transportable agents should be able to easily access external resources that provide learning and reasoning capabilities.

## 2.3 Distributed information retrieval

Here an *agent* is a program that searches multiple information resources for the answer to a user query. Typically the resources contain large volumes of data and are distributed across a network of heterogeneous machines. In addition the agents are characterized by (1) the use of knowledge-intensive techniques to avoid brute-force search and extensive manual intervention and in some cases (2) the concurrent execution of multiple subsearches and *communication of partial results from one subsearch to the others*. The partial results from one subsearch are used to narrow the scope of other subsearches.

Etzioni and Weld present a softbot-based interface to the Internet which accepts a request and then develops a plan that satisfies the request using available Internet resources. Softbot stands for "software robot" and comes from the *artificial intelligence* definition of an agent – i.e. an agent is an entity that acts on the environment with effectors and perceives the environment with sensors. In this case the sensors are Internet resources such as archie, gopher and netfind. The effectors are resources such as ftp, telnet and mail. The softbot uses declarative logic to encode knowledge about the available Internet resources and about how to map requests into resource invocations. The softbot accepts requests that are expressed in a subset of first-order logic and performs a standard backtracking search to develop an appropriate plan. The request can be anything as long as the necessary knowledge is encoded in the knowledge base. The example in the paper uses a combination of Internet resources to resolve underspecified e-mail addresses when sending messages.

Vesser has written a succession of papers that develop a model for distributed searching. The most recent is [OPL94] which recasts the model in terms of agents. In the model a search involving multiple distributed resources is performed by a collection of cooperating agents. There is one agent for each resource. Each agent is responsible for searching its assigned resource and for communicating partial search results to the other agents. The partial results are used to narrow the scope of each search to more relevant sections of the resource. The standard example is a vacation planner that searches multiple information resources – a weather database, a car rental database, a hotel database and a "places of interest" database – in order to plan an appropriate vacation for the user. There is one agent in charge of searching each database. The agents must communicate since there are complex interdependencies between the information in each database – i.e. the information discovered in one database could invalidate the information discovered in another. For example the beach is a poor "place of interest" if there are thunderstorms forecast. At the same time the agents must not deadlock with each waiting for a piece of information that one of the others must provide. Therefore each agent assumes enough information so that it can proceed completely independently if necessary. For example the place agent might assume good weather. Partial results are communicated from agent to agent in order to refocus the searches. Eventually the agents arrive at a consistent vacation plan.

These examples and this definition of an agent do not demand a particular implementation. Indeed the search could run entirely at a single site and simply invoke the necessary remote services. However transportable agents provide a natural and efficient method of implementing such a search. A transportable agent could

be sent to the site of each relevant resource. Each agent is responsible for searching the resources at its site and for communicating partial search results to the other transportable agents.

## 2.4 Software interoperation

Here an *agent* is a program that communicates correctly in a *universal* communication language. An agent can interoperate with any other agent – even if they have different underlying implementations – since they use the same communication language. This definition of an agent is closely related to agent-based software engineering in which applications are implemented as a collection of autonomous, cooperating peers. There are two approachs to agent-based interoperation – *procedural* and *declarative*.

### 2.4.1 Procedural

In this approach agents exchange *procedural directives*. The recipient executes the procedure in order to perform some task on behalf of the sender. Most existings systems that use the procedural approach are based around high-level scripting languages. Applications are sent scripts that guide the application through the desired task. These applications are said to be *scriptable* or *programmable*. Notable examples of the script-based approach include Tcl, Telescript, Apple Script, Hewlett-Packard's NewWave environment and the Autonomous Knowledge Agents (AKA) project [GK94, Joh93]. The Tk extension to Tcl provides the *send* command which is used to send Tcl scripts from one application to another. Telescript allows agents to send scripts to each other once the agents have established a direct connection. Apple Script and NewWave provide similar scripting functionality. The main goal of the AKA project is to "develop an architecture for creating and running personalized software agents". However a key component of the system is a uniform interface for all electronic resources. Each electronic resource such as WAIS and FTP is surrounded with a Tcl interface. Agents communicate with the resource using the procedures in the Tcl interface rather than the resource's native communication language. This makes agents much easier to write. In addition the Tcl interface can remain constant across different architectures which makes agents much easier to port.

### 2.4.2 Declarative

Genesereth [GK94] points out several disadvantages of the procedural approach. Writing procedures might require information about the recipient that is not available to the sender; procedures only compute in one direction; and procedures are difficult to merge. Genesereth argues for a declarative approach in which agents exchange *declarative statements*. The recipient performs an inference process in order to derive results from the sender's declarative statements. These declarative statements are written in the Agent Communication Language (ACL). ACL has three components – a vocabulary, an inner language called the Knowledge Interchange Format (KIF) and an outer language called the Knowledge Query and Manipulation Language (KQML) [GK94]. The vocabulary is a dictionary of words specific to the application area. Each word has an English definition and a set of formal annotations written in KIF. KIF is a prefix version of first order predicate calculus which can express data, relationships among the data and procedures or scripts. The atoms of KIF are the words from the vocabulary. A KQML expression consists of a directive followed by one or more KIF expressions. Directives include *telling* an agent that a KIF expression is true, *asking* an agent if a KIF expression is true and so on. KQML offers more efficient communication than KIF alone.

Agents that use KQML can communicate with each other directly. However this places the burden of interoperation squarely on the programmer. Instead Genesereth proposes a federated architecture in which *facilitators* handle interoperation [GSS94]. Essentially each agent is assigned a facilitator. An agent communicates only with its facilitator but facilitators communicate with each other. Each agent posts its *capabilities* and application-specific facts to its facilitator. *Capabilities* are a description of the services that the agent provides. Capabilities and facts are expressed in KIF. When an agent needs information, it sends a request to its facilitator. The facilitator uses backward inference to find an answer to the request. The inference process might involve decomposing the request into subrequests and invoking other agents to handle each subrequest. The advantage of the facilitator approach is that each agent communicates with a a single

system agent that *appears* to handle all requests itself. The main concern with the facilitator approach is scalability – i.e. the size of the shared vocabulary, the cost of the inference process and the size of the facilitator's knowledge base. The first problem is addressed by allowing agents to use different vocabularies and providing translation features; the second and third problems are addressed by limiting the amount and kind of information that each facilitator stores internally.

The federated approach is similar to directory assistance, distributed object managers and automatic brokers. Directory assistance allows a program to find a desired service. Distributed object managers provide transparent access to a distributed collection of objects. Messages are automatically routed to the destination object even if the sender does not know the object's network location. Automatic brokers provide both functions by first identifying an appropriate recipient for a message and then forwarding the message. An example of directory assistance is the X.500 protocol; distributed object managers include CORBA, DSOM, OLE and OpenDoc; automatic brokers include ToolTalk and the Publish and Subscribe Service on the Macintosh [GK94]. The federated approach is distinguished by the amount of processing done in the facilitator [GK94]. Each facilitator performs backward inference rather than simple pattern matching.

### 2.4.3 Procedural versus declarative

It is unclear whether the procedural or declarative approach is better. The procedural approach is suggested when the sender is requesting a task that the recipient does not know how to perform in its entirety. The declarative approach is suggested for knowledge-intensive applications in which assertions need to be exchanged or in which planning and inference are required. Transportable agents represent a hybrid approach. The transportable agent itself is a procedure that migrates to a remote machine so that it can execute at the location of the data. However transportable agents do not have to communicate with procedural directives. The communication facilities are flexible enough to allow any communication protocol including the exchange of declarative statements. In addition the agent can make use of external services that provide planning and inference capabilities. There is no need to build declarative logic into the agent language.

One of these approaches to software interoperation – directory assistance, distributed object managers, automatic brokers or federated inference engines – will be essential in a transportable agent system so that an agent can (1) find agents that perform a needed task and (2) communicate with agents without knowing their current network location. This is particularly essential since transportable agents are *mobile*. Keeping track of recipient locations without system support would be a nearly intractable programming challenge even if all the agents came from the same application and communicated only among themselves. Whichever solution is adopted, it must be extended so that it performs effectively in a highly dynamic environment. Transportable agents come into existence, change network location and terminate continuously. These changes must be visible to other agents.

## 2.5 Transportable agents

Transportable agents are the focus of the thesis. A transportable agent is a named program that can migrate from machine to machine in a heterogeneous network. The program chooses when and where to migrate. It can suspend its execution at an arbitrary point, transport to another machine and resume execution on the new machine. First we discuss message passing, remote procedure calls and remote evaluation which are the forerunners of transportable agents. Then we describe existing transportable agent systems.

### 2.5.1 Message passing and remote procedure call

The message passing model provides two communication primitives. *send* transmits a message to a destination process while *receive* accepts an incoming message. Message passing can be used in client/server computation. The client sends a request message to the server. The server handles the request and sends a reply message to the client. The reply message contains the result of the request. The send and receive primitives can be *blocking* or *nonblocking* and can be *synchronous* or *asynchronous*. *Blocking* means that the

primitives do not return control to the caller until the message has been successfully sent or received; *non-blocking* means that the primitives return immediately. *Synchronous* means that the send primitive does not return until the recipient issues a corresponding receive; *asynchronous* means that the send primitive returns as soon as the message arrives on the destination machine. The destination machine buffers the message until the recipient issues a receive. Message passing is powerful and flexible but requires the programmer to handle low-level details – keeping track of which response goes with which request, converting data between client and server formats, determining the address of the server and handling communication and system errors [SS94].

Remote procedure call (RPC) [BN84] was designed to relieve the programmer of these details. RPC allows a program on the client to invoke a procedure on the server *using the standard procedure call mechanism.* The most common implementation – which is the implementation of [BN84] – uses *stub* procedures [SS94]. A client that makes a remote procedure call is actually calling a local stub. This client stub puts the procedure name and parameters into a message and sends the message to the remote machine. A server stub on the remote machine receives the message, extracts the procedure name and parameters and invokes the appropriate procedure. The server stub waits for the procedure to finish and then sends a message containing the result to the client stub. The client stub returns the result to the client. The original RPC mechanism blocked the client until the server returned the result. Most extensions to traditional RPC allow concurrent invocation of the same procedure on multiple servers or make RPC asynchronous [SS94]. These variations are more flexible but make programming more difficult.

Other disadvantages of traditional RPC were described in [GG88]. It is difficult to send incremental results from the server to the client; implementations are commonly optimized for short results rather than bulk data transfer; and there is no way to pass a procedure – or more precisely a reference to a procedure – to the server. This last limitation obviates any protocol that requires the server to invoke a client-specified procedure *on the client machine.* [GG88] addresses these problems by allowing procedure references to be passed as arguments and by introducing the *pipe* abstraction. A *pipe* is a connection between the client and the remote procedure that exists for the duration of the remote procedure call. Incremental results and bulk data are transferred along the pipe. There are additional stubs at either end of the pipe to handle the data transfer.

### 2.5.2 Remote evaluation

A minor problem with RPC is that a distributed application must be written as two distinct pieces – i.e. the client and the remote procedures. This is a step backwards from a *precursor* of RPC. Hamlin's programming system at the University of North Carolina was designed for graphics applications. The programmer wrote a single program which the compiler divided into interactive and computation-intensive pieces. The interactive piece was executed at the graphics terminal while the computation-intensive piece was executed on the mainframe. The two pieces were tied together with stubs as in RPC [Spr95]. The critical problem with RPC is that the client is limited to the operations provided at the server. Since it is unlikely that the server provides an operation that meets the client's needs exactly, the client must make several remote procedure calls, bringing intermediate data across the network on every call. If the intermediate data is not useful beyond the end of the client's task, a significant amount of network bandwidth has been wasted. To address these two problems, researchers have turned to remote evaluation in which a *subprogram* is sent from the client to the server. The subprogram executes on the server and returns its result to the client.

Falcone [Fal87] describes a system in which clients and servers program each other using the Network Command Language (NCL). NCL is a variant of Lisp. Each server provides a library of NCL functions. A client that requires a service sends an NCL expression to the appropriate server. The expression can use any functions provided at the server or sent as part of the expression. The server evaluates the expression and returns the result to the client. The result is an NCL expression itself and can perform arbitrarily complex processing on the client. The important point is that an NCL expression can contain functions that act as control structures. Thus the NCL expression can invoke multiple functions at the server and avoid the overhead of multiple remote procedure calls. The prototype system provides implicit access to servers by translating systems calls into appropriate service requests. First the system identifies the appropriate server

to handle the system call; the system call is translated into an NCL expression; the NCL expression is sent to the server and evaluated; the server returns an NCL expression which is evaluated on the client to produce the return value of the system call.

Remote evaluation (REV) is similar to NCL in that a procedure can be sent to a remote server for evaluation [SG90]. However REV can be used with any language. It uses client and server stubs in much the same way as RPC. All that is needed for a new language is stub generators and linking facilities so that the procedure can invoke the procedures provided at the server. The procedure can be transmitted as source, intermediate or compiled code. The choice of transmission format depends on the language, the desired level of security and the heterogeneity of the network. Stamos and Gifford identify four security considerations – authentication, availability, secrecy and integrity. Authentication and availability consist of checking the identity of the client and preventing malicious denial of service. These can be handled with well known techniques [SG90]. Secrecy and integrity consist of preventing unauthorized access to and destruction of server information. These are more difficult. Stamos and Gifford present three solutions – separate address spaces for each procedure, careful interpretation in a single address space and digital signatures with a single address space. The first and third support compiled code while the second avoids the overhead of multiple address spaces. Digital signatures is an open research area in which a program is compiled by a trusted third party. The third party checks the program for security violations and applies a cryptographic signature to the compiled code if no security violations are present. The server knows that certain security checks have been performed already if it receives a digitally signed procedure. The advantage of REV over NCL is that REV can be incorporated into any programming language which allows the programmer to use the most appropriate language for the application. On the other hand NCL is symmetric in that procedures can be sent from the client to the server *and* from the server to the client.

REV and NCL impose significant restrictions on the procedures that can be sent to the remote machine. The main limitation is that the procedure must be self-contained. All functions and variables referenced in the procedure must be provided at the server or sent along with the procedure. This means that the semantics of a procedure call are different for remote calls than for local calls - i.e. a remote procedure can not access anything that is external to the passed subprogram. The developers of REV and NCL were primarily concerned with moving computation to a remote machine and imposed this limitation to simplify implementation. SURPRA-RPC (SUbprogram PaRAmeters in Remote Procedure Calls) on the other hand seeks to allow normal procedure call semantics for both local and remote calls [Sto94]. Essentially SUPRA-RPC extends traditional RPC with additional stubs that are invoked whenever a procedure references an out-of-scope variable or function. The server makes a callback to the client in order to handle the out-of-scope reference. SUPRA-RPC implementations exist for C, C++ and Lisp although the C and C++ implementations work only in a homogeneous environment since compiled code is passed from machine to machine.

### 2.5.3 Transportable agents

Transportable agents extend the remote evaluation schemes by allowing autonomous programs to migrate from machine to machine. This approach is a convenient paradigm for distributed applications and simplifies the task of communicating among distributed subprograms. Table 1 summarizes the existing transportable agent systems. Some of these systems fit the definition better than the others in that they actually allow a program to suspend its execution *at an arbitrary point* and resume execution on a new machine. The others allow subprograms to be sent to a remote site but not the internal state of *executing* subprograms.

There are several schemes that fall between remote evaluation and transportable agents. Postscript programs are often sent to remote printers and displays. Scripting systems such as Apple Script allow scripts to be sent from one application to another [Joh93]. MIME/Safe-Tcl allows Tcl scripts to be embedded in e-mail messages. These scripts are executed automatically when the message is received or viewed [BR93]. The IBM Intelligent Communications Network$^{TM}$ uses Intelligent Objects$^{TM}$. These objects can contain procedures and can be sent from an application on one site to an application on another. The recipient can execute the embedded procedures [Rei94]. Oracle Mobile Agents is meant for mobile computing and associates an agent with each mobile user. However the agent is not transportable. It resides in the central network and acts as

| System | Language | Language features | Suspendable | Communication | Security | Crash Recovery |
|---|---|---|---|---|---|---|
| Jodler | Tcl (extended) | Imperative Interpreted Object-oriented | No | Invocation of object members | Authentication Access/clearance Cost/allowance | No |
| Obliq | Custom | Imperative Interpreted Object-oriented | No | Invocation of object members | Protected objects Lexical scope | No |
| TACOMA | Tcl | Imperative Interpreted | No | *meet* | UNIX | Rear guards |
| Routers | Custom | Expression Interpreted | Yes | Yes | Unknown | ISIS |
| Kotay and Kotz | Custom | Imperative Interpreted | Yes | No | Access lists | No |
| Telescript | Custom | Imperative Interpreted Object-oriented | Yes | *meet* *connect* | Credentials Permits | Backing store |
| Agent Tcl | Tcl | Imperative Interpreted | Yes | *send* *receive* | No | No |

Table 1: A comparison of existing transportable agent systems – Agent Tcl is the system that is being implemented as part of this thesis. It is far from complete.

the user representative when the user is disconnected from the network or when the user does not want to receive intermediate data due to low transmission rates and high latencies [ora94]. Some of these schemes do not support arbitrary distribution and migration. The rest are application-specific and ill-suited to general computation. These schemes are not included in the comparison of existing transportable agent systems.

*SodaBot* – an agent environment and construction system – was developed to facilitate the creation of *personal assistant* agents [Coe94]. The system provides a high-level scripting language, a *basic software agent* (BSA) that controls the execution of all agents associated with a particular user and a graphical interface to the BSA. Users can change the access rights for each agent running on their BSA and can visually inspect agents for security threats. SodaBot supports limited migration in that the components of an agent application are automatically distributed to the appropriate network sites when the application is first used [Coe94]. However this is far removed from true migration. SodaBot is not included in the comparison of existing transportable agent systems.

*Jodler* is an object-oriented extension to the Tool Command Language (Tcl) [Ott94]. which object in Jodler consists of named slots which can contain either functions or values. A value can be a reference to another object. The *transfer* primitive moves an object to a new network location. A member function executes at the location of its object. Thus an application can create an object, transfer the object to a remote site and invoke the object's member function in order to perform computation on the remote site. Jodler incorporates two security mechanisms. *Access/clearance* associates a clearance level with each object and an access level with each slot. An object can access a slot only if its clearance level is higher than the slot's access level. *Cost/allowance* associates an allowance with each object and a cost with each slot. When an object accesses a slot, its allowance is decremented by the slot's cost. Access is denied if the cost is greater than the object's remaining allowance. The weakness of Jodler is that *no* distributed version has been implemented. In other words there is no version in which objects can actually migrate. The author states that this is because he has been unable to develop an object identification scheme that works across multiple Tcl interpreters – i.e. across multiple network sites. This is a puzzling statement since there are a wide range of successful approaches to distributed object management. Until an identification scheme is developed and the migration feature is implemented, Jodler will be just an object-oriented extension to Tcl with rudimentary security features.

*Obliq* is similar to Jodler except that it has a fully realized – and more elegant – implementation [Car94]. Obliq is an interpreted, lexically scoped and object-oriented. An Obliq object is a collection of named fields which contain methods, aliases and values. A value can be a network reference. Obliq objects are technically immobile and tied to a specific site. However an object can be created at a remote site, cloned onto a remote site or effectively migrated with a combination of cloning and redirection. In the latter case a copy of the object is created at the remote site and all references to the original are redirected to the copy. An Obliq program – just like a Jodler program – can create an object at a remote site and perform remote computation through member invocation. However Obliq extends Jodler in two significant ways. First the Obliq system provides a name server. Objects register their location and a descriptive name with the server. Objects use the server to locate other objects. Second the Obliq system allows procedures to be migrated in addition to objects. A procedure can be sent to a generic *compute engine* on the remote site. The compute engine executes the procedure and returns the result. The procedure can contain unbound variables which are variables defined within the lexical scope of the procedure but not in the procedure itself. These unbound variables are automatically replaced on the remote machine with network references to the local copies. Thus Obliq remains lexically scoped even when a computation is spread across multiple sites. In addition to making programs easier to reason about, lexical scoping is an important security mechanism. A procedure can access only those objects defined in its scope or given to it as procedure arguments. This means that it is impossible for a procedure to access resources on a remote machine without explicit cooperation from the server. The second security feature of the Obliq language itself is that objects can prevent other objects from cloning and modifying them. Higher-level security measures are not addressed.

TACOMA (Tromsø and COrnell Moving Agents) is a mobile agent system that uses Tcl/Horus. [JvRS95]. Tcl/Horus is a version of the Tcl scripting language which uses Horus to provide group communication and fault-tolerance. TACOMA agents are written in Tcl. Each agent has a *briefcase* of *folders*. Folders contain Tcl data and procedures. The single abstraction is the *meet* operation which an agent uses to execute another agent. The *meet* operation passes a briefcase to the invoked agent. The briefcase contains the information that the agent needs to perform its task. All services except for *meet* are provided directly by other agents. For example an agent executes a Tcl script on a remote machine by sending the script to the *rexec* agent. An agent can migrate by sending *itself* to the rexec agent. However TACOMA does not support the interruption of executing Tcl scripts. As in Jodler and Obliq the migrated script does not continue from the point of interruption. Instead it is executed from the beginning. This lack of true migration makes it difficult to implement certain distributed applications but not impossible since state information can be explicitly stored in the data that accompanies the script. Important features of the TACOMA system include rear guards, electronic cash and brokers. Rear guard agents handle machine failures. A rear guard is left behind whenever an agent migrates to a new machine. The rear guard restarts the agent if the agent suddenly "vanishes" [JvRS95]. Electronic cash is used to pay for services and is intended as a security measure. Runaway agents are impossible since an agent can not continue once its finances are exhausted. TACOMA relies on the underlying UNIX system for additional security. Broker agents provide directory and scheduling services.

Intelligent routers support true migration - i.e. the migration of *executing* scripts [WVF89]. An intelligent router is a software analogy for the run card that travels with a manufacturing lot. A run card specifies the steps that need to be taken in order to correctly manufacture the final product as well as the results of previous steps. Intelligent routers are written in MP1 and MP2 which are interpreted, expression languages. Routers are evaluated by an interpreter at each site. A router can request migration to a new site in which case the current state of the router is captured and sent to the new machine. The router resumes execution on the new machine from the point of interruption. Each machine has a dispatcher that accepts the incoming routers. Routers written in MP2 can communicate with each other, spawn child routers and catch exceptions. MP1 is simpler and does not provide these facilities. A version of the router system that runs on homogeneous machines uses ISIS to detect and recover from node failures and other faults [WVF89]. However there is no corresponding fault-tolerant implementation for heterogeneous machines. There apparently is a mechanism for detecting malicious routers but this mechanism is not specified. In addition the current status of the router research is unclear. However intelligent routers have been used in a wafer routing application [Voo91] and have been suggested for a range of workflow applications [WVF89] in which certain people or machines must perform certain steps of a task. Intelligent routers are a natural mechanism for workflow applications since they can physically move from machine and machine and person to person as each step in the task

is completed. The router carries data relevant to the task along with it and performs arbitrarily complex processing when deciding which step to perform next.

Kotay and Kotz [KK94] describe a prototype system that was implemented as a starting point for future research. Transportable agents are written in a simple scripting language that includes most of the features of AWK as well as "traditional imperative language features" such as *if-then* and *while* [KK94]. An agent migrates with the *moveto* command. The *moveto* command captures the internal state of the agent and transports this state to a new machine. The agent continues execution on the new machine from the statement after the *moveto*. Three transport mechanisms were tested – electronic mail, UNIX remote shell and TCP/IP. The authors note that the actual transport mechanism is unimportant since the agent cares only that it has changed location rather than how it arrived. Security is provided with *access lists* which specify the external programs that an agent is allowed to invoke on the remote machine. The system is missing important components but it was intended as a testbed rather than for industrial-strength applications. It has been used to search distributed collections of technical reports. The authors suggest six areas for future work – an interagent communication mechanism, replacement of the custom scripting language with a standard scripting language such as Tcl, an agent reproduction mechanism, improved security, improved error handling and navigation issues such as accessing replicated resources and searching for relevant resources [KK94].

Telescript is a General Magic product that is being used in AT&T's new PersonaLink network. It is one of the most robust transportable agent systems although it is a commercial product and unfortunately is not available to researchers. Each network site is divided into one or more virtual places. Telescript agents are written in an object-oriented scripting language. An agent uses the *go* command to migrate to a new place. The agent continues executing at the new place from the statement after the *go*. Agents can interact with other agents in two ways. An agent can *meet* with an agent that is in the same place. The two agents receive references to each other and communicate using standard object-oriented programming techniques. In addition an agent can *connect* to a remote agent. The agents pass objects along the connection. Each network site runs a Telescript engine or server that maintains the places at the site and executes incoming agents. The Telescript engine continuously writes the internal state of executing agents to nonvolatile store so that the agents can be restored after a node failure. In addition the Telescript engines enforce two security mechanisms. Each agent carries cryptographic credentials. Each place checks these credentials in order to authenticate the agent's identity. In addition each agent has certain *permits* which are renegotiated whenever the agent enters a new place. Permits give an agent the right to use a certain Telescript instruction or to use a certain amount of an electronic resource. For example a permit could specify a maximum agent lifetime or a maximum fiscal budget. Agents that attempt to violate the conditions of their permits are terminated immediately [Whi94]. The current applications of Telescript are electronic shopping (where each place is the electronic representative of a sales organization), active e-mail (where each place is a mailbox or router) and certain network management tasks [Rei94].

### 2.5.4 Distributed systems

Transportable agents can spread across a network and cooperate to perform a task. These cooperating agents behave like a distributed system. Therefore many of the services that will be necessary or convenient in agent applications have been addressed in distributed systems research. These services include synchronization, mutual exclusion, deadlock detection, agreement, shared memory, scheduling, transactions, crash recovery, fault tolerance and reliable broadcast [SS94]. These services must be available to transportable agents. However it is unreasonable to provide all of these services at the system level. Such a system would be unwieldy and inefficient. Instead most of these services should be provided with other *agents*. The challenge is to identify those services that must be incorporated into the system and those that can be *efficiently* provided with other agents. Then the existing solutions can be copied and pieced together into a complete suite of services.

### 2.5.5 Weaknesses of existing systems

The main weakness of existing transportable agent systems – except for the systems that do not support arbitrary migration – is that they focus on certain technical challenges to the exclusion of others. For example Telescript has impressive security and low efficiency. The second weakness of existing work is that there has been no formal characterization of agent performance.

# 3 Proposal

This these addresses these two weaknesses by formally characterizing the performance of transportable agents and implementing a complete system.

## 3.1 Performance modeling

There has been no formal characterization of agent performance. Instead researchers and developers *assume* that transportable agents consume fewer network resources than the client/server models. Evidence for this assumption is either anecdotal or missing. Although the assumption clearly holds for specific applications, the range of applications for which it holds is unknown. This means that it is difficult to choose a communication paradigm for a distributed application. And since the relationship between performance and network characteristics is unknown, it is difficult for an agent to select an appropriate communication strategy in the face of changing and unexpected network conditions. For example a transportable agent could limit itself to the equivalent of remote procedure calls if (1) bandwidth is high, (2) latency and the chance of network disconnection are low and (3) servers charge less for remote procedure calls since extensive security checks are not required. The difficulty lies in determining the exact point at which it becomes cost effective to switch from migration to remote procedure calls.

Therefore the first phase of the thesis is to determine the network, data and application parameters under which transportable agents offer the best performance. A transportable agent can engage in four basic communication behaviors.

1. Naive – The agent downloads data to its current location. This can be viewed as a special case of remote procedure call in which each call is a blind request for data.
2. Remote procedure call – The agent makes a call across the network in order to invoke a remote service.
3. Remote evaluation – The agent sends a child agent to a remote machine. The child agent executes and returns its result. The child agent can send out children of its own.
4. Migration – The agent migrates from machine to machine.

A simple representation of the four models is shown in figure 2. *Naive* involves a single request to the server and then a bulk transfer of all data. *Remote procedure call* involves a sequence of requests and responses. *Remote evaluation* involves the transmission of a child agent and its result. *Migration* involves transmission of the agent from site to site. A transportable agent can limit itself to any of these communication behaviors. There are two essential performance measures.

1. Traffic – the total network traffic produced by a distributed application
2. Compute cycles – the total number of compute cycles consumed by a distributed application (on both local and remote machines)

In addition a wide range of cost functions can be defined. Three cost functions are immediately apparent.

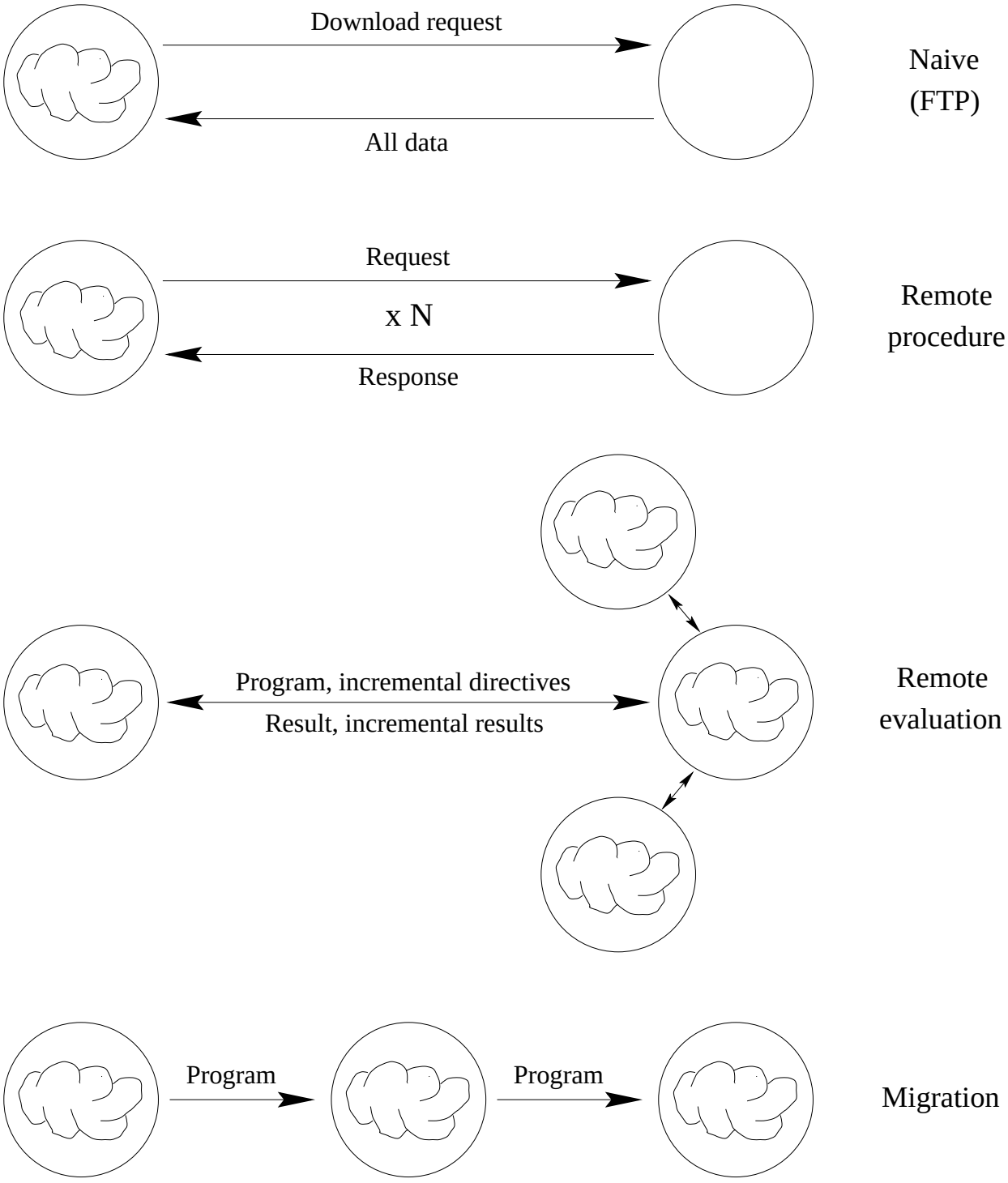



Figure 2: The four communication behaviors of interest – naive, remote procedure, remote evaluation and migration. The cloud-shaped objects indicate where custom client computation can be performed.

1. Latency – the total time that the distributed application takes to complete its task (in isolation and in the presence of competing applications)

2. Money – the total financial cost incurred by a distributed application. This is an important cost function since it is likely that remote machines will charge for certain services.

3. Latency and money – the combination of latency and financial cost

There are three tasks for which a performance model will be used – predict the performance of a given agent, select the agent that can most efficiently perform a given task and select the network that can most efficiently support a collection of agents. Efficiency can be measured in terms of any cost function.

### 3.1.1 Predict agent performance

Here the task is to predict the performance of a well-known agent given the current network conditions. It is assumed that we have an exact description of the agent's behavior. This assumption will be relaxed in subsequent sections. The parameters that must be considered in making such a prediction include

1. the topology of the network

2. the bandwidth and latency of the network links

3. the current load on the network links

4. the financial cost associated with each service

5. the exact description of the agent's behavior. This includes the path that the agent takes as it migrates through the network, the size of the agent's code section and the reduction in this code section between successive migrations, the processing time at each network site, when and where the agent spawns children, when and where the agent communicates with local or remote peer agents, when and where the agent interacts with a local service, when and where the agent invokes a remote service and pulls data across the network, when and where the agent blindly downloads data, and how much data is carried along with the agent on each migration. Each child agent has its own behavior and there is a size associated with each message or data item that is passed across the network. Broadly speaking a description of an agent's behavior specifies how the agent combines the four basic communication behaviors in order to accomplish its task.

Under our assumption that an agent can precisely describe its behavior none of these parameters are difficult to model. The parameter that will be the most difficult to *obtain* in a real network is the current load on the network links. It clear that we can not ask one of the sites connected to the link for the link's current load. This would significantly increase the load on all the links in the network. In addition the load could fluctuate rapidly in which case the load figure would be inaccurate by the time that it was returned to the home site and passed into the performance model. It seems clear the we must predict the expected load on the basis of past observation or settle for the load figures that are passed from adjacent site to adjacent site anyways for use in routing algorithms.

### 3.1.2 Select an agent

Here the task is to select the agent that can most efficiently accomplish the given task *or more precisely* for an agent to select the most efficient communication strategy. It is assumed that we have a precise description of the agent's task. This assumption will be relaxed in subsequent sections. Efficiency can be measured in terms of any cost function. The communication strategy includes all of the items that made up the description of an agent's behavior in the previous section. An agent can exert at least indirect control over all these items and direct control over most. In addition to the parameters from the previous section we must now consider

1. Distribution of resources across the network

2. Data relevance. This specifies how much data at each resource is relevant to the agent's task and how much of this relevant data must be transferred off site.

3. Data redundancy. This includes both the presence of replicated resources and the number of duplicates when items obtained from two different resources are merged. its search.

4. Dependencies. This includes whether certain steps must be executed at certain sites or against certain resources *and* whether certain steps can be executed in parallel. These dependencies determine whether it is worthwhile to spawn child agents to perform subtasks, whether these child agents need to communicate partial results and whether the agent can choose from one of a number of functionally equivalent resources.

5. Existing agents and services. This specifies whether there are existing agents and services that the agent can use to accomplish part of its task. In effect this is a measure of the complexity of the task since a complex task is less likely to be implemented as a third party service. If the third party service is not available, the agent must perform the processing itself and can not possibly make a call across the network. It must either download the data or migrate.

Again none of these parameters are hard to model since we assume that we have a precise description of the task. However we should model expected rather than exact relevance and redundancy since in a real application it is unreasonable to assume that we know a priori how much data at each resource is relevant or duplicated. In a boolean search application we might model expected relevance on the basis of the size of the collection and the expected number of documents per term. Such predictions could be provided by the resource on demand or could be cached at the home site if network latency is high relative to the processing time at the resource. The redundancy and relevance parameters can be incorporated into the prediction model from the previous section to relax the assumption that the agent's behavior is known precisely. The control strategy of the agent remains known but now the amount of data that it obtains at each step is nondeterministic. This leads to a more general prediction of performance.

It is worthwhile to examine a concrete example of strategy selection. Consider an agent whose task is to search a distributed collection of resources. The resources contain generic items. The user of the agent specifies a query against which items are ranked and a threshold, The agent must find all items whose rank is greater than the threshold. There are two extremes. The agent can migrate from resource to resource or the agent can spawn one child per resource. The children search the resources and return the retrieved items to the parent. These extremes are specific cases of two general *partitioning* strategies. The agent can partition the network sites and spawn one child agent per partition. Each child migrates sequentially through the network sites in its partition. Alternatively the agent can partition the network sites and proceed sequentially through the partitions. At each partition the agent spawns one child agent for each network site in the partition. The agent proceeds to the next partition only when the children have finished. The agent's task is to choose the partitioning strategy, the partitions and either the ordering of the partitions or the ordering of the sites within each partition. The agent's choice should minimize some cost function. We will consider both latency and cost. To ease the discussion, assume a simplified model in which the only relevant parameters are network bandwidth and the number of items obtained from each site. To make the discussion nontrivial, assume that the resources contain redundant items. The agent expects that d items will be above the threshold at each site and that $d * f(n)$ of these items will be unique where n is the number of sites searched.

Latency is the time that it takes the agent to complete its task. It might appear reasonable to minimize latency by sending one agent to each site. However this is reasonable only when the communication links have infinite capacity. Consider the case of performing a search with a personal communicator in a moving car. The connection to the network is low-bandwidth wireless and is often unusable due to noise or obstruction. If f(n) is $n^{1/2}$, sending an agent to each of four sites brings twice as much data across the wireless connection as sending a single agent that migrates sequentially through the sites and throws out duplicates. If the transmission dominates the search time, sending one agent to each site would take nearly twice as long. The

same argument holds for any slow or heavily loaded link especially when one considers that there could be thousands of agents making the same decision. Sending one agent to each of five machines at the University of Melbourne in Australia is a bad idea for a user at Dartmouth. If an agent can spawn children regardless of its current network site, it is of course most efficient for the agents to fan out through the network and eliminate redundant items as they fan back in. However it is likely that many machines will impose strict limits on the depth and degree of the child hierarchy in order to prevent virus-like behavior. Under these conditions the agent must partition the network and send one migrating agent to each partition. The partitions must be chosen so that the time gained from parallel execution is exactly balanced with the time gained due to lower network traffic. It is conceivable that this partitioning problem is NP-complete. However there are several approximations such as clustering the network sites in order of decreasing link performance. The clustering process stops when the next union would cause an increase in latency. The agent must be able to determine the *current relationship* between the effective transmission rate of a link and the amount of data that it wants to send across the link.

Resources can charge for the services that they provide. In this case the arguments developed for latency become even more persuasive. For example if a resource charges for each item that is obtained from the resource, financial cost is minimized by sending out a single agent that migrates sequentially through the sites. The agent carries previously discovered items along with it so that it can recognize redundant items. At each site the agent obtains and is charged for new items only. If a machine charges for each byte that it receives or transmits on behalf of an agent *or* if a network link charges for each byte that is transferred across it, the cost of retrieving redundant items must be balanced against the cost of carrying previously seen items along with the agent. Here an agent can define the financial cost of a link as the cost of transferring data along the link plus the cost of bringing data onto the destination machine minus the cost of being unable to check for redundant items. The cost of being unable to check for redundant items would include the cost of eventually transmitting a redundant item to the home machine. Then the agent can proceed as in the latency case. The agent must be able to determine the price of a service *before* it uses the service. Combining latency with finances produces a more complicated optimization problem. Adding back the remainder of the parameters makes matters even worse.

### 3.1.3 Select a network

Four distinct tasks are collected under this heading – predict the performance of a class of agents, select an agent communication strategy for a class of applications, predict the performance of a network that is being subjected to agent traffic, and select an appropriate network architecture for agent traffic. The common thread in all of these tasks is that it must be possible to model the behavior of a *typical* agent. There is no longer a precise description of the task or of the agent's behavior. In other words we need to model the typical machine access pattern or the typical resource usage pattern. Modeling of *typical* access patterns has been one of the most difficult aspects of machine and network simulation [Spr95]. If the model is too general, model results will not agree with actual performance. If the model is tuned to particular applications, model results will agree with actual performance only for a particular mix of applications. One possible approach is to identify typical applications and then characterize the behavior of each application. However it is conceivable that agents will fall neatly into categories according to their dominant communication behavior. For example some agents will primarily migrate from site to site while others spawn a hierarchy of children. This suggests that we can use the four basic communication behaviors as categories and characterize the behavior of a typical agent from each category. The characterization could be made in terms of Markov decision processes. For example the Markov decision process for migrating agents might indicate that these agents rarely reproduce. Simpler characterizations could be possible. For example it is not unreasonable to suggest that agents that invoke *remote* services extensively will migrate so seldomly that is is not worth considering. Similarly it might be reasonable to model the migration behavior of migrating agents as a simple rate. The models for typical agents must be refined in parallel with application development so that we can examine and capture the behavior of actual agents. At the same time we will not be implementing a large number of applications so we will need to make intelligent assumptions along the way. An effective model for *typical* agents will allow the consideration of network architectures and will allow the prediction of agent behavior and the selection of agent strategy in much more general situations.

### 3.1.4 Modeling strategy

It is likely that the relationships between the parameters and agent performance are complex enough to make mathematical analysis nearly intractable. Instead network simulation and experimental verification will be used to determine the relationships numerically. Then we must explicitly state the model such that an agent can select parameters on the basis of desired performance levels. How the model will be expressed depends on what relationships are suggested through the simulation work. Hopefully mathematical relationships, rules and algorithmic procedures will become apparent for common cases. Alternatively simulation results can be memorized. An agent will use a nearest-neighbor scheme to select the simulation result that is closest to the current network conditions and the desired performance. The parameters for that simulation result are the parameters for the agent. In parallel the agent can run a simulation for the current conditions and add a new point to the memorized set. It is unreasonable to run the simulation and then perform the task since the latency will be severe and the observed network conditions will change by the time that the simulator finishes and the agent begins executing. Other automated learning techniques such as neural networks will be explored as necessary if we are unable to obtain effective prediction. The final task is to use the model to select appropriate networks. This is the most complicated task and could reduce to running multiple simulations with careful enumeration of possible network topologies – i.e. we do not want to enumerate all of them. The critical concern with selecting a network is modeling a *typical* agent. A concern with any use of the model is the timely delivery of current network and resource conditions to the machine on which the model is being used. Delivery in a few specific cases was mentioned above and will be considered throughout the implementation phase.

## 3.2 Implementation

A transportable agent system must support two basic tasks – the migration of an agent from one network site to another and communication between distributed agents. In addition the system should be efficient, fault tolerance, flexible, private and secure. Existing transportable agent systems do not meet all of these criteria. Even the commercial language Telescript focuses on some technical challenges to the exclusion of others. Therefore the second phase of the thesis is to implement a *complete* transportable agent system. A large portion of this work will involve identifying and extending *existing solutions* in the distributed computing, operating system, programming language, artificial intelligence and transportable agent literature.

The components that must be implemented are listed below. The list is divided into two broad categories – components whose functionality requires support at the system level and components whose functionality can be provided in agents running on top of the system. The division of components is preliminary and could change as the components are integrated into a coherent whole. In addition there is no review of the existing research that is relevant to each component. The amount of relevant research is so large and scattered that it is beyond both the time frame and scope of this thesis proposal. Research into each component will occur throughout the course of the thesis as each component is developed. Here we simply highlight the approaches taken in existing transportable agent systems.

### 3.2.1 System level

1. MIGRATION. A transportable agent can suspend its execution at an arbitrary point, migrate to a new machine and resume execution on the new machine. This requires support at two levels. The programming language must provide a mechanism for capturing the internal state of an executing agent. In addition there must exist an entity that can transmit agents to other sites and execute agents arriving from other sites. All existing systems that support true migration have a server on each machine that sends and receives agents. Agents are written in a custom scripting language that is designed to allow the capture of internal state. The scripts are interpreted rather than compiled for security and portability reasons. A special language primitive such as the *go* instruction in Telescript captures the internal state of an executing script and passes the state to the local server for transmission to the destination server. A similar migration mechanism must be implemented in our transportable

agent system. However we seek to extend a standard scripting language such as Tcl rather than design a custom language.

2. COMMUNICATION Agents must be able to communicate among themselves. The first issue is the level of the communication primitives. Low-level primitives such as *send* and *receive* are flexible but impose a significant burden on the programmer [SS94]. High-level primitives are restrictive but easier to use. The existing transportable agent systems provide either implicit communication or high-level primitives such as *meet*. For example communication in Obliq requires nothing more than the invocation of a member function. The member function can belong to a local or remote object. TACOMA and Telescript use the *meet* primitive which contacts the recipient and returns when the recipient has explicitly accepted or rejected the meeting. This is safer but less flexible than send and receive. We seek to provide both low and high-level primitives. It should be possible to build the high-level primitives on top of the low-level primitives with careful implementation. The second issue is maintaining communication with agents that are continuously changing network location. This can be partially supported with a low-level *forwarding* scheme although efficient support will demand one of the interoperation techniques that were discussed in related work.

3. INTERACTION WITH ELECTRONIC RESOURCES. An agent must be able to access the electronic resources that are available at each machine. There are three possibilities. The agent can interact with the resource using the resource's native communication mechanism; the resource can provide a library of procedures that can be called from inside an agent; or the resource can be encapsulated within an agent. The first approach is unsafe and nonportable. The second approach is attractive for resources such as filesystems. The third approach involves more communication overhead but provides a uniform interface across all resources – i.e. accessing a resource means communicating with the agent that controls the resource. The third approach can be subdivided into transducers, wrappers and rewrites. A transducer is an agent that communicates with the resource in the resource's native communication language. Transducers work with any resource but involve more communication overhead. An alternative approach is to place a wrapper around the application that effectively turns the application into an agent by redefining its interface. This approach involves less communication overhead but the source code of the application must be available. The third approach is to rewrite the application which is the most time-consuming but allows optimization for an agent environment [GK94]. We seek to support both procedure libraries and agent encapsulation. Agent encapsulation means that an agent must be able to *directly* access any resource at the site. However only the agents that encapsulate the resources will actually be allowed to do so.

4. SECURITY. There are two aspects of security. First a resource must be protected against malicious or badly programmed agents. These malicious agents might access restricted information or damage the resource. All existing transportable agent systems use interpreted languages so that there is an additional layer between the agents and the bare hardware. The existing systems provide different degrees of security on top of the interpreted language. Telescript provides the most complete security. All Telescript agents have permits and credentials. Credentials are used to continuously authenticate the identity of an agent's owner while permits limit the scope of an agent's actions and resource usage [Whi94]. The Telescript security mechanisms appear to be sufficient. We seek to confirm their sufficiency and either copy or redesign as appropriate.

   The second and more subtle security issue is that an agent must be protected from a malicious machine or resource. This issue has been mentioned extensively in the literature but has seen no implementation work or proposed solutions. It is clear that it is impossible to protect an agent from the machine on which the agent is executing. It is equally clear that it is impossible to protect an agent from a resource that willfully provides false information. However it is essential that there be a way to detect whether an agent has been modified inappropriately by a previous machine when it migrates to a new machine. Otherwise the new machine might assign a security violation to the *owner* of the agent. The owner would then be subject to undeserved sanctions. We seek to implement a verification mechanism so that each machine can check whether an agent was modified unexpectedly after it left the home machine. Such a verification mechanism might involve communication with the home machine.

5. PRIVACY. A transportable agent might contain information about its owner in order to make appropriate decisions. A third party can infer information about the agent's owner by examining this internal information or simply by observing the external behavior of the agent. It is impossible to prevent a machine from examining an agent that is executing under its control. Instead we must hide the identity of the agent's owner while allowing the service to perform security checks and bill for services rendered. Different services will allow different levels of privacy. Providing privacy is trivial if there is an independent third party that is trusted by both the service provider and the agent's owner. For example the AT&T PersonaLink network is currently organized around a centralized AT&T server that is implicitly trusted. Therefore the server can submit an agent to a service on behalf of an anonymous user. The service sends billing information to the server which forwards it to the actual user [Rei94]. Privacy becomes much more difficult and perhaps impossible if there is no trusted third party. Fortunately high-privacy applications are often low-security while high-security applications are often low-privacy – e.g. browsing an online store catalog versus accessing classified military information. This should greatly extend the range of potential solutions. We seek to provide as much privacy as possible while maintaining billing and security.

6. EFFICIENCY. One of the key factors that will determine the success or failure of a transportable agent system is the efficiency with which an agent can carry out its assigned task. Efficiency issues arise in every phase of the implementation. However three issues warrant special attention – accessing replicated resources, caching the results of previous agents in case the next agent wants to perform the same task, and efficiently executing agent code. It is unclear how much can de done at the system level in terms of accessing replicated resources and caching results. The problem with replicated resources is how to redirect an agent to a lightly loaded copy when it is explicitly requesting migration to a specific copy at a specific network site. The problem with caching is how to detect when an agent or an agent fragment will produce a result that was recently produced by another agent. Agents are written in a general-purpose scripting language so detecting functionally identical agents is extremely difficult unless we consider only those agents that are lexically identical as well. Support for replicated resources might have to be moved to the agent level where an agent can query a resource manager about the copies that are available and which copy will provide the fastest turnaround. Caching might have to be provided by each resource individually or in the application itself.

   Efficiently executing the agent code can be done no where but the system level. Existing systems use interpreted languages for security and portability. However the interpretative overhead is severe. For example Tcl runs ten thousand times slower than native C [SBD94]. The clear solution is to either compile agents into a low level representation that can be interpreted much faster or to compile agents into actual machine code. The Java$^{TM}$ programming environment takes both approaches. A high-level Java program is compiled into bytecodes. The bytecodes can be interpreted or translated into machine code. Interpreted bytecodes are much faster than Tcl. Bytecodes converted to machine code are nearly as fast as native C [jav94]. Alternatively several groups are working on faster Tcl interpreters [Sah94] and Tcl compilers [SBD94]. These groups have achieved notable success even though Tcl was never meant to be compiled. There are other reasonable choices for an agent language. The tradeoff is that security mechanisms become more complicated as the agent gets closer to the bare hardware. It would not be unreasonable for a site to charge more for a compiled agent than an interpreted one or to disallow compiled agents altogether. Therefore the challenges are to (1) select a language that supports interpretation *and* compilation, (2) allow the agent to request compilation once it reaches the destination machine and (3) meet security requirements even if the agent is compiled. It is not critical which language is used in the thesis work as long as it supports both interpretation and compilation. We plan to use Tcl as discussed in the next section. The important questions are how much the system would need to change if we switched to a different language and whether the system can support multiple languages at the same time. A different language should require nothing more than an appropriate interpreter, compiler and security module. The rest of the system should remain unchanged. Multiple languages will require a uniform means of identifying the language of an incoming agent. A compiled agent must be compiled on the destination machine since it is unreasonable to expect a machine to produce correct machine code for every potential destination machine. In addition the destination machine can perform as much security checking during the compilation process as desired.

We feel that the best approach is a two phase scheme. First a distribution agent migrates to each machine, compiles the code and registers the code with a special-purpose agent. Then the application agent migrates from machine to machine using the precompiled code as needed. Such an approach allows the detection of compiler errors before the application begins running.

7. DATA TYPES The basic data types that transportable agents use must be identified and incorporated directly into the language. In addition it must be possible to access arbitrarily complex data structures that are defined externally to the agent language – i.e. data structures whose operations are implemented in some other language with an interface for transportable agent use. For example transportable agents that learn might need neural networks. The challenge with external data structures is transmitting their internal state and their implementation to the remote machine.

8. NETWORK AND RESOURCE AWARENESS. A transportable agent should be able to determine the characteristics of the current machine and of the network between the current machine and any destination machine. Relevant characteristics include capacity, latency and load. The agent can make processing and communication decisions on the basis of these characteristics. For example if the destination machine is connected over a low-bandwidth wireless network, the agent might choose to transmit image titles rather than image thumbnails. Similarly the agent should be able to determine the status of any resource or agent.

9. CROSS PLATFORM. Transportable agents are specifically meant for a heterogeneous environment. Therefore we must ensure that the transportable agent system will work on arbitrary platforms across arbitrary communication channels. In practical terms this means that we can not limit the implementation to Unix workstations since it is not difficult to port programs from Unix to Unix and all of the workstations use the same communication protocol (TCP/IP). The most reasonable choice of a non-Unix platform at Dartmouth is a Macintosh communicating over an AppleTalk network. Here the key issue is allowing migration between the Unix and AppleTalk networks while maintaining transparent communication. The agent should be unaware of the communication mechanisms even if it has to cross the network boundary. Fortunately the underlying network hardware and software should handle most of this task. Potentially the single consideration in the agent system itself will be to provide an appropriate naming scheme so that an agent can refer to any connected machine *with a high level name*.

10. DEVELOPMENT ENVIRONMENT. Transportable agents – like all distributed paradigms – demand careful programming. A robust development environment is essential in order to ease the programming task. It is not the intention of this thesis to implement a production-quality development environment. However three development tools will be implemented. First there is a need for a flexible debugging facility that can track transportable agents as they migrate through the network. Second most users will not have the desire or the proficiency to write a transportable agent. Instead most applications will translate high-level user actions into low-level transportable agents. For example an information retrieval application would translate a query and a list of potentially relevant resources into a transportable agent that migrates from resource to resource in order to answer the query. We seek a toolkit of functions that will simplify the task of turning a high-level representation into a transportable agent. Such a toolkit can be used directly in an application or can be used as the basis for an application-specific toolkit. Third it has been suggested that it should be possible to modify an agent *while the agent is executing*. [WVF89]. It is unclear whether the ability to modify an agent is useful in the general case. However it is certainly useful in network management and workflow applications. Workflow applications require that certain people or machines perform certain steps of the task. The person responsible for each step or the steps themselves can change in midstream. It must be possible to modify the agent that is directing the task without restarting the task from scratch. However this modification is accomplished, it must be noted that there is no need for arbitrary modification. It is more a matter of replacing a well-defined piece of the agent while the agent is suspended or while independent pieces continue executing.

The following components are critical in a robust, flexible system but have seen extensive development in

the context of distributed systems. We expect to use the existing solutions rather than develop novel ones. The challenge is to integrate the existing solutions with the other system components.

1. CRASH RECOVERY The transportable agents executing on a node should not be lost forever if the node crashes. Instead it is desirable to perform as much automatic recovery as possible. TACOMA leaves a rear guard agent behind whenever an agent migrates to a new network site [JvRS95]. The rear guard is responsible for restarting the agent if the agent disappears. The rear guard scheme quickly becomes complex since an agent that migrates several times will lead to chains or cycles of rear guards. Telescript takes an alternative approach and continuously backs up the internal state of an executing agent to nonvolatile store [Whi94]. All agents can be restored from the nonvolatile store if the site crashes. An issue that does not seem to be considered in the TACOMA or Telescript implementation is that there might be other agents that depend on one of the agents on the crashed machine. These agents must wait until the machine comes back, find an alternative resource or terminate. The system must at least notify these agents that the machine has crashed so that they can take the appropriate action. Ideally the system would provide an *option* to freeze a dependent agent when a machine crashes and unfreeze the agent when the machine comes back up. However this scheme quickly becomes complex since the dependents of the dependents must be frozen and so on. A related issue is that if an application assumes that a child agent on a crashed machine is gone forever and takes alternative action, the child agent should remain gone. It should not come back when the machine comes back.

2. FAULT TOLERANCE Many other faults can occur besides site crashes. At the system level we seek to detect these faults and take simple corrective action – e.g. a single retry. Transportable agents will be notified of the faults if the corrective action fails.

3. SYNCHRONIZATION. Agents that cooperate to perform some task must be able to synchronize their actions. Agents should be able to synchronize whether they are local to one machine or distributed across multiple machines. At the system level we are concerned with low level mechanisms such as barrier synchronization.

4. INTERRUPTS. It should be possible to asynchronously notify a transportable agent that some event has occurred. This is similar to standard operating system interrupts except that we would expect the events to be at a much higher semantic level. There are two challenges – (1) provide the language primitives that establish interrupt handlers and tell the system which events should generate interrupts and (2) allow interrupts to cross the network. It should be possible for a transportable agent to receive an interrupt whenever an event occurs *on some other machine*.

5. DEADLOCK. The potential exists for a collection of agents to deadlock. This is particularly true if an agent can hold exclusive access to a resource across multiple program instructions or if an agent can block for an arbitrarily long period of time while waiting for communication from some other agent. Deadlocks must be detected and broken. It is possible that deadlock handling will fall out of the security mechanisms and no dedicated deadlock detection mechanism will be required. For example if each agent has a maximum *wall clock* lifetime, deadlocks are guaranteed to be broken after sufficient time has elapsed. Each resource might specify a maximum time that an agent can hold mutually exclusive access to the resource. More complex schemes that are required in specific applications might be implemented at the agent rather than system level.

### 3.2.2 Agent level

The agent level is concerned with services that can be provided with other agents. Here we present some of the services that will be critical in certain applications. The focus of the thesis is to ensure that the facilities at the system level allow efficient implementation of the services at the agent level. As part of this work we will implement at least a simple version of each service. Each of these services has been researched extensively. Again this research is not summarized here. A key aspect of the work will be surveying the existing approaches and determining the ease with which they can be implemented on top of the system.

1. COORDINATION. This includes high level mechanisms such as process group tools, simulated shared memory and so on.

2. FAULT TOLERANCE AND CRASH RECOVERY. This includes high level mechanisms such as transactions and reliable transmission of the same message to distributed agents.

3. LEARNING Some agents must learn how the user performs a task or must improve their future performance based on past experience. Such learning involves indirect observation of the user and other agents as well as direct feedback.

4. PLANNING AND INFERENCE Some agents must plan how to accomplish their task.

5. FINANCES. Transportable agents may need to pay for certain services. TACOMA [JvRS95] uses a trusted money server that assigns a large random integer to each piece of electronic cash. When an agent transfers electronic cash to another agent, the recipient verifies that the cash has not been spent by contacting the authentication server. The server assigns a new random integer to the cash so that the recipient can spend the money that it has just received. There is a clear window of opportunity for a malicious agent or agents to spend the same piece of cash over and over again in the time that it takes the first recipient to contact the money server. TACOMA allows agents that feel cheated to request an audit from an impartial third party. Therefore each TACOMA agent must document its actions if it wants to take part in the audit process. TACOMA's electronic cash has the important security benefit of preventing runaway agents. An agent can not continue onces its finances are exhausted. PersonaLink is a commercial network that is centered around billing [Rei94]. However the exact billing mechanism is unknown.

6. NEGOTIATION Transportable agents may need to negotiate with one or more other agents in order to settle on a satisfactory price for the service.

7. RESOURCE DESCRIPTION, ORGANIZATION AND DISCOVERY. These issues arise in information retrieval applications. Each resource should have a high level description that summarizes the contents of the resource. Transportable agents can examine this description in order to make a broad determination as to the resources relevance or irrelevance. This would allow the agent to immediately eliminate some resources from consideration. In addition resources should be organized into hierarchies, lattices or some other structure where groups of summaries are summarized themselves. The agent could then eliminate entire groups of resources from consideration. Finally an agent must be able to efficiently search collections of resource descriptions in order to identify previously unknown but relevant resources.

8. SCHEDULING. Scheduling is the task of selecting a machine for an agent when there are several machines that can satisfy the agent's needs. A scheduling mechanism will be critical in supporting access to replicated resources.

9. INTERFACE TO RPC AND REMOTE EVALUATION SERVERS. There should be auxiliary agents that provide interfaces to existing remote procedure and remote evaluation systems. This would allow transportable agents to easily access traditional servers.

## 3.3 Performance evaluation

The final phase of the thesis is to build several applications on top of the transportable agent system and evaluate system performance. The applications should exercise all components of the system and therefore should cover the range of agent behavior. There are three basic agent behaviors. An agent can communicate with resources and agents across the network; an agent can migrate from site to site; or an agent can spawn child agents and coalesce the child results. Most applications exhibit more than one of these behaviors but often one behavior dominates. We have not determined the exact applications that will be implemented. However we seek to build one or more applications from each of four categories. Each category focuses on a different behavior.

1. DISTRIBUTED INFORMATION RETRIEVAL. Here we intend to focus on search applications in which agents fan out through the network and then fan back in. The agents coalesce and summarize their results as they fan back in. Three search domains are under consideration – two dimensional images, mechanical parts and medical records. Prototypes of the medical record and mechanical parts applications have been implemented on top of the prototype agent system. The prototype system is described in the next section. In addition to agent reproduction these applications will be a test of (1) agent compilation since the queries will involve mathematically intensive comparison and redundancy metrics for which compiled speed is essential, (2) resource description, organization and discovery since an agent should be able to discover previously unknown resources, (3) scheduling since an agent should be sent to a lightly loaded copy of a replicated resource and (4) caching since search results from previous agents should be stored and reused.

2. PERSONAL ASSISTANTS. A personal assistant relieves the user of a routine, burdensome task. Here we plan to focus on interagent communication and interaction with resources. In other words the personal assistants should achieve their goals using the services of existing agents and resources. Three personal assistants are under consideration. All three have information retrieval aspects as well. The first personal assistant watches the user's calendar and to-do list and searches the Internet for information relevant to each item. The hope is that the assistant will identify useful information between the time that the user adds an item and the time that the user begins to address the item. The second personal assistant accepts a stream of information – e.g. a news feed – and throws away the items in which the user is not interested and searches for additional information on the items in which the user is extremely interested. The job of this assistant can be viewed as creating a personalized newspaper or news broadcast for the user. The third personal assistant is perhaps the most ambitious and most entertaining. We seek to implement part of Nicholas Negroponte's digital house [Neg95] in which smart appliances communicate with each other in order to accomplish a task for the owner. For example it is easy to imagine an assistant that coordinates with the alarm clock, hot water heater, coffee pot and microwave (into which a dish of oatmeal has been placed the night before) and with an external taxi service in order to ensure that the owner gets up, cleans up and makes it to the airport in time for his flight. Most importantly this assistant should adjust every component accordingly when the airline notifies it that the flight has been delayed. In addition to interagent communication and resource interaction these applications will be a test of (1) the learning capabilities of an agent since it must improve its future performance based on past experience, (2) the planning and inference capabilities of an agent since it must identify possible solutions to the task and select the best solution according to some criteria, (3) finances and negotiation since it is likely that the agent will need to pay for certain services, (4) interfaces to traditional servers since it is unlikely that the agent system is available at each Internet site, (5) the privacy mechanisms that guard a user's identity and (6) the mechanisms that allow an agent to determine resource status since resource could be unavailable or heavily loaded. The latter four are less applicable to the digital house so it is unlikely that the digital house will be the primary application.

3. WORKFLOW. A workflow agent provides the overall control for a task in which certain people or machines must perform certain steps. The agent migrates from site to site and person to person according to who is responsible for the current step. The processing to decide which step is performed next can be as complex as desired. Three workflow applications are under consideration. The first is procurement in which the task is to identify a needed part or service, identify the organizations that can provide the part or service, solicit bids from these organizations, select the winning bid and order the service or part from the winner. Different people are responsible for the procurement process at different points. For example the initiator submits the initial request, the comptroller approves the expenditure and the auditor reviews the bid selection process. The second application is robotic manufacturing. Here we imagine that a company produces custom parts on a robotic manufacturing line. The robots must be told what steps to take for each part. As the part moves along the assembly line and passes through each manufacturing tool, an agent migrates through the corresponding controllers. The agent guides the tool through the appropriate steps for the current part, records the results and handles exceptional conditions. This application is an extended version of the wafer routing application in

[Voo91]. The third application is software development. Here an agent guides a software package and its components through the software life cycle. In addition to migration these applications are a test of (1) fault tolerance and crash recovery since losing a workflow agent in midstream would be catastrophic, (2) coordination since there might be one agent for each parallel track of the task and (3) the mechanisms that allow an agent to determine the status of other agents since it must be possible to examine the task progress.

4. ACTIVE E-MAIL OR DOCUMENTS. Active e-mail allows each message to include one or more scripts. The scripts are executed automatically when the message is received or viewed. Active documents are similar. The scripts embedded in active documents are executed when the document is received or viewed or when a portion of the document is selected. Here the focus is to demonstrate the ease with which active e-mail and documents can be implemented on top of the agent system since these applications are seeing extensive discussion in the literature.

All of these applications will be a test of the security precautions, the basic data types that are available to the agent and the toolkit that converts high-level user actions into actual transportable agents. There is no point in developing production-quality interfaces for every application however. Most of the interface work will be concentrated on the personal assistants. In addition I have associated each system component with the application for which the component seems to be most critical. This is not to suggest that the component is unimportant in the other applications. For example resource discovery might be critical for the first two personal assistants. Most large applications will exercise a significant portion of the system.

These applications and the system itself will be instrumented to obtain exact performance figures. Performance figures include measures such as load, latency and fiscal cost. The observations will be used to verify the accuracy of the models that were discussed in the *modeling* section. The models must be refined if they are inconsistent with actual performance. Once an accurate model is developed, a transportable agent can use the model to select an appropriate communication and migration strategy. Using the model allows the agent to adapt to changing and unexpected network conditions. To test strategy selection, we plan to modify the applications listed above. Each of these applications was associated with a particular control strategy in order to fully exercise the system. However although these control strategies are a reasonable choice for these applications, they are not the most efficient for certain network and agent characteristics. The guts of the applications will be replaced with code that obtains the current network, resource and agent characteristics and then combines these characteristics with the model in order to select the best control strategy. As noted above we do not want each agent to implement its own version of the model. Instead there should be a generic third party that suggests good strategies based on network conditions and as good a description of the task as the agent can provide.

The instrumentation will be a documented part of the system so that it can be used in future work. Of course it will be possible to turn off the instrumentation in order to avoid measurement overhead.

# 4 Status

## 4.1 Performance modeling

No work on performance modeling has been done so far. However we are in the process of identifying existing network simulation tools. We expect to use the architecture simulation language CARL which is an extension of the discrete event simulator PARSIM. CARL was designed for the simulation of CPU internals but can be used equally well for computer networks.

## 4.2 Implementation

We have implemented a prototype system that uses the Tool Command Language (Tcl) as the transportable agent language. The architecture of the system is shown in figure 3 and described in detail below. The casual reader may wish to skip to the final paragraph in this section.

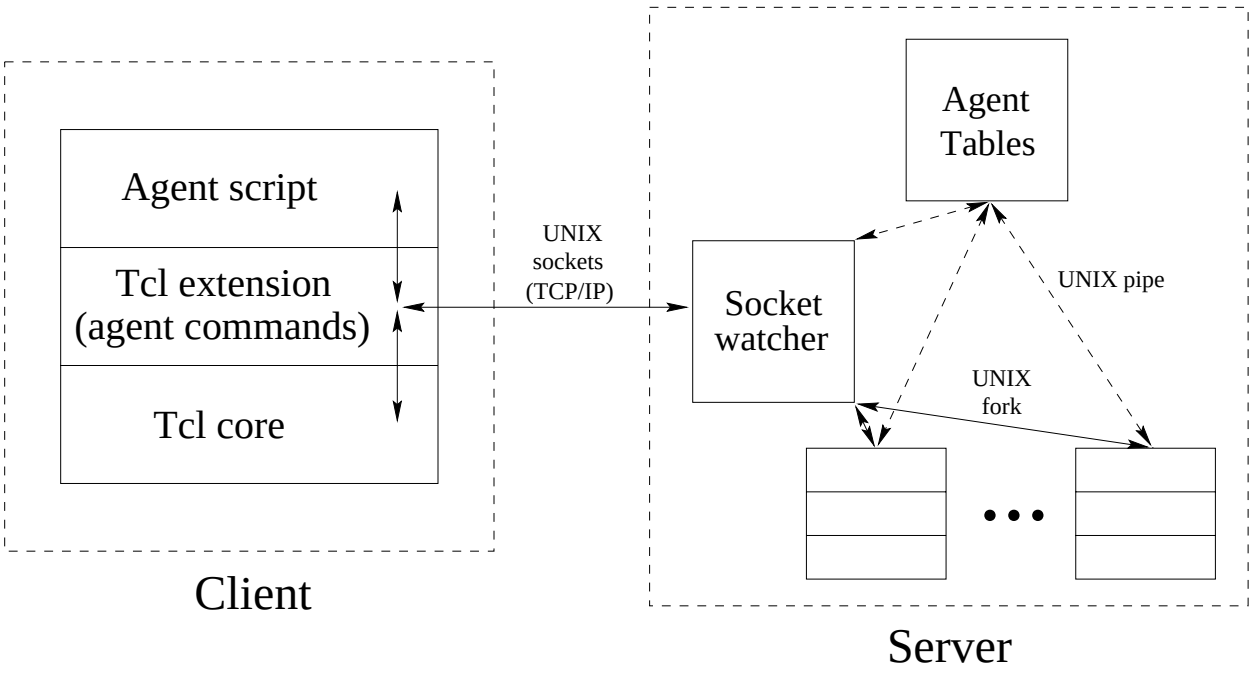



Figure 3: The prototype transportable agent system

Tcl is a high-level scripting language that was developed by Dr. John Ousterhout at the University of California at Berkeley in 1987 and has enjoyed enormous popularity since then. Tcl has several advantages as a transportable agent language [Ous94]. Tcl is interpreted. This makes Tcl scripts highly portable and simplifies the implementation of security precautions. Tcl can be *embedded* in other applications due to its dual existence as a stand-alone interpreter and a function library. This makes it easy for applications to implement *part* of their functionality with transportable agents. Tcl can be extended with user-defined commands. This allows each information resource to provide a package of Tcl commands that can be used to access the resource. Agents dynamically load the command package and then invoke the desired commands. Such dynamic loading of resource-specific commands is a lower level but more efficient alternative than encapsulating the resource within a transportable agent. Finally Tcl is freely available to researchers unlike the commercial language Telescript. Several existing systems use Tcl as the agent language as noted in related work.

However Tcl has several disadvantages. Tcl is inefficient compared to other interpreted languages such as Perl [Ous95]. Tcl is *not* object-oriented and provides no code modularization except for procedures. This makes it difficult to write large scripts. Fortunately several groups are working on more efficient Tcl interpreters [Sah94] and on Tcl compilers [SBD94]. In addition the lack of object-oriented features has not been an issue with the Tcl agents developed so far since these agents have been small even though they perform significant processing at the remote site. However it will become an issue as script size increases. In this case there are several existing object-oriented extensions to Tcl. It will be interesting to see if a transportable agent can use the object-oriented extensions without object-specific support from the underlying system. This should be possible if each extension provides procedures to save and load its internal state. The system can call these procedures during state capture and restoration without any knowledge of the extension's purpose. This assumes that the extension is available on both the source and destination machines. Some existing systems use the object-oriented extensions to Tcl but provide object-specific support in the underlying system.

Capturing the internal state of Tcl extensions is future work. First we need to capture the internal state of ordinary Tcl scripts. Unfortunately the final disadvantage of Tcl is that it provides no facilities for capturing the internal state of an executing script. Adding such facilities turned out to be relatively straightforward but required the modification of the Tcl core. The essential problem is that the Tcl interpreter evaluates a script

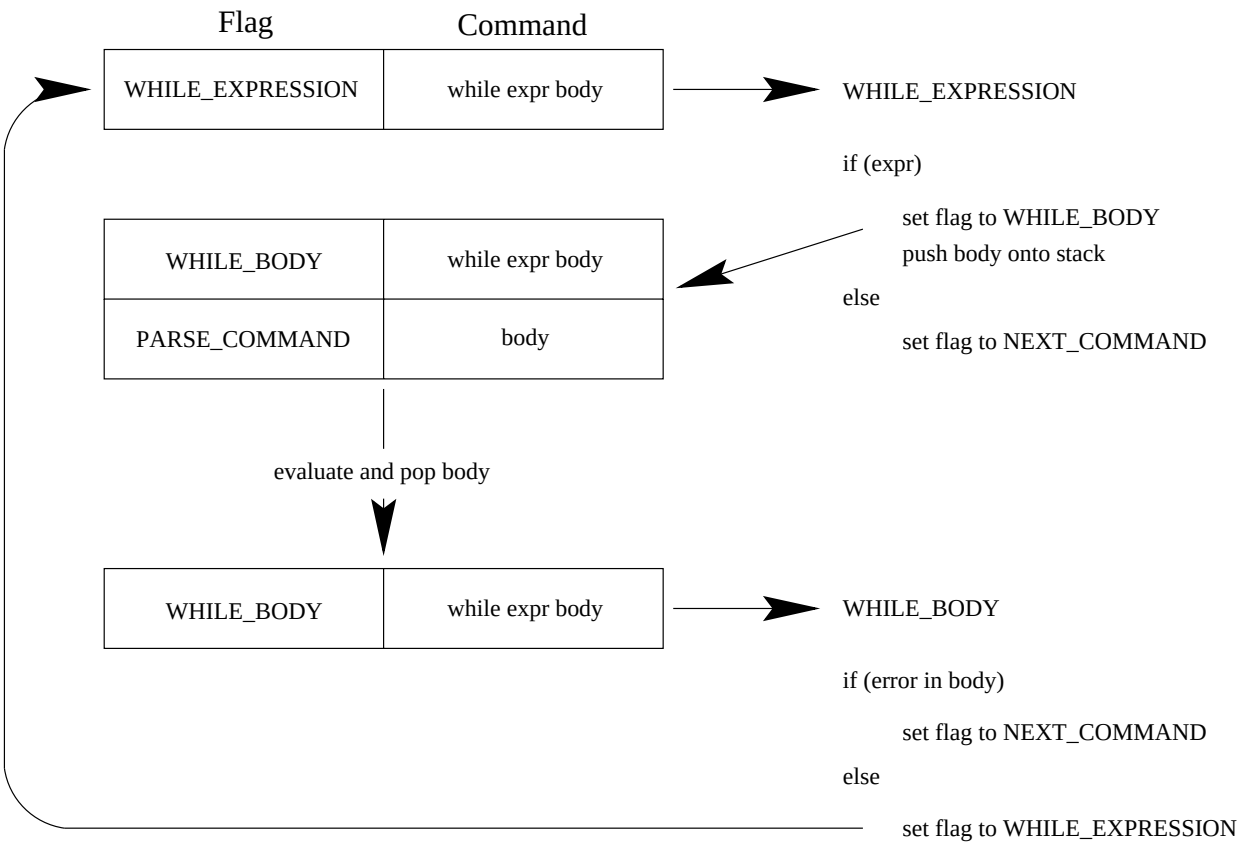



Figure 4: The interaction between the explicit stack and the two handlers for the *while* command

by making *recursive* calls to the main evaluation procedure Tcl_Eval. This means that the most important part of the script's state – namely its command stack – is implicitly stored in the interpreter's stack to which there is no easy access. The solution was to add an explicit stack to the Tcl core and replace each recursive call to Tcl_Eval with a push onto the stack. Tcl_Eval was expanded with a loop that iterates until the stack is empty. On each iteration Tcl_Eval calls the evaluation procedure associated with the command *on the top of the stack*. The command is popped off the stack once it has been evaluated.

We have implemented a prototype system that uses the Tool Command Language (Tcl) as the transportable agent language. The architecture of the system There are two complications with this simple scheme. The first complication is that some command procedures perform command-specific processing both before and after the recursive call. An example is *while* procedure which recursively calls Tcl_Eval to evaluate the body of the while loop. The solution was to divide these procedures into two or more new procedures. Each procedure contains the code that previously appeared between two successive calls to Tcl_Eval. The *while* procedure was divided into a procedure that evaluates the control expression and a procedure that handlers error conditions in the body. A state flag is associated with each command on the stack. The state flag specifies which of the new procedures is the current handler for the command. Each new procedure sets the state flag as it finishes. For example the procedure that evaluates the control expression of a while loop either sets the state flag to *next command* or pushes the body of the loop onto the stack and sets the state flag to *while body*. In the second case Tcl_Eval evaluates the body, returns to the while loop on the stack and calls the procedure that handles error conditions in the body. This procedure sets the state flag to either *next command* or *while expression* depending on whether or not an error occurred. In the second case Tcl_Eval calls the procedure that evaluates the control expression and the sequence repeats. A command is popped off the stack as soon as its flag is set to *next command*. This process is illustrated in figure 4. Careful choice of state flags meant that just those procedures that involve a recursive call to Tcl_Eval had to be touched.

The second complication is that the programmer might wish to capture the script state in the middle of a command substitution. The problem is that command substitutions are evaluated while the command is being *parsed* which made it nearly impossible to eliminate the recursive call to Tcl_Eval. Thus we kept the recursive call but added a special variable table into which is stored the value of each command substitution. When the state is captured during a command substitution, the table is captured as well. When the state is restored, the interpreter parses the *entire* command again but checks the table whenever it performs a command substitution. If the command substitution has already been performed, it uses the value in the table rather than evaluating the substitution again.

Once the explicit command stack was added to the core, it was trivial to write procedures that save and restore the state of an executing Tcl script. These procedures save and restore the stack, the variable tables, the procedure tables, the commands in the script, the procedure call frames and certain global parameters. These procedures are the heart of the transportable agent system.

The transportable agent system has two components. The first component is a server that runs on each machine to which transportable agents can be sent. The server is implemented as two cooperating processes. The *socket* process watches the network connection and accepts incoming agents. The *table* agent keeps track of the transportable agents that are executing on its machine and buffers messages that have been sent to an agent but not received. The second component of the system is the modified Tcl core as described above and a Tcl extension that provides agent commands. The most important commands are agent_submit, agent_fork, agent_jump, agent_send and agent_receive. These commands are used to move the agent through the network and to send and receive messages. Transportable agents are written in standard Tcl except that they must be evaluated with an interpreter that uses the *modified Tcl core* and includes the extension.

agent_submit sends a Tcl script to a remote machine for evaluation. The agent_submit command connects to the *socket* process on the remote machine and sends the script over the connection. The *socket* process creates a child process to handle the script. The child communicates with the *table* process in order to obtain a unique local identifier for the agent. Then the child creates a Tcl interpreter, evaluates the agent and notifies the *table* process when the agent terminates or migrates.

agent_submit sends a complete Tcl script which is executed from the beginning. agent_fork and agent_jump send the internal state of an executing script. These two commands capture the internal state of the script, connect to the *socket* process and send the internal state over the connection. The *socket* process creates a child process which obtains a unique local identifier from the *table* process, creates a Tcl interpreter, *loads the state image into the interpreter* and continues evaluating the agent from the point at which it was interrupted. agent_jump terminates the original script – i.e. the agent is *migrated* to the remote machine. agent_fork does not terminate the original script – i.e. the agent is *cloned* onto the remote machine. The *table* process is notified when the agent terminates or migrates.

agent_send and agent_receive provide a rudimentary message service. agent_send sends a message to a destination agent. The destination agent is specified by providing the network location of the agent and its local identifier. This location-dependent addressing is easy to implement but makes it difficult to communicate with an agent that is migrating from machine to machine. agent_send connects to the *socket* process on the destination machine. The *socket* accepts the message and immediately passes the message to the *table* process. The *table* process holds the message until the destination agent issues an agent_receive command. agent_receive communicates with the *table* process in order to obtain the most recent message. Successive calls to agent_receive obtain successively older messages.

An important feature of the message service is automatic result passing. The result of a Tcl script is the result of the last command executed in the script. When an agent finishes executing, the script result is automatically sent to the *top level* agent as a message. The *top level* agent is the agent at the root of the hierarchy created by the submit, fork and jump commands. The top level agent can receive the result just like any other message. Figure 5 shows a sample transportable agent that illustrates these commands. The agent submits a second agent. This second agent jumps from machine to machine and executes the Unix "who" command at each machine. The list of users on each machine is concatenated onto a growing list. When the second agent finishes, the list is automatically sent to the first agent as a message. The first agent receives the list and displays it to the user. The use of two agents is a side effect of a current technical

```
set num 4
set machine(1) {muir.cs.dartmouth.edu}
set machine(2) {tenaya.cs.dartmouth.edu}
set machine(3) {tioga.cs.dartmouth.edu}
set machine(4) {tuolomne.cs.dartmouth.edu}

agent_begin tioga

  # submit the script that will jump from machine to machine.  We send the
  # scalar variable "num" and the array "machine" since the script needs
  # these two variables. Note the msg variable in the submitted script.  At
  # each machine we append the who list for that machine to msg.

agent_submit tioga -vars num machine  \
    -script {

       set msg ""

       for {set i 1} {$i <= $num} {incr i} {
         agent_jump [set machine($i)]
         set who_info [exec who]
         set location $agent(local)
         append msg "$location:\n$who_info\n"
       }

         # We want to return the concatenated who lists so we just do
         # a "return $msg".  This is the last command in the submitted
         # script so the script result ends up being the value of msg.
         # The script  result is automatically returned to the submitter
         # as a message.

       return $msg
}

  # get the result from the submitted script

set result -1

while {$result == -1} {
  set result [agent_receive msg]
}

puts $msg
agent_end
```

Figure 5: A sample transportable agent that executes the "who" command on multiple machines

limitation. When an agent migrates, it loses its connection to the terminal device and can not regain this connection even when it jumps back to the original machine.

The prototype system implements only two of the components that were discussed in the proposal – migration and rudimentary communication – and has several implementation weaknesses that must be addressed – lack of a blocking or timeout facility for the agent commands, extraneous copying of messages to and from the *table* process and loss of terminal connection. However the system has been used successfully to search distributed collections of medical records and mechanical parts. The transportable agents were easy to write and Tcl was more than efficient enough for the searching task. Because of this initial success, Tcl and the existing prototype system will be used as the starting point for the remainder of the thesis work.

### 4.3 Performance evaluation

No work on performance evaluation has been done so far since we have just finished the prototype system. However application development is underway. Two information retrieval applications are being developed on top of the system. One application is concerned with the retrieval of medical records while the other is concerned with the retrieval of mechanical parts. There are working prototypes of both of these applications.

## 5 Conclusion

Transportable agents are a more efficient means of accessing remote resources than traditional client/server models. However existing research into transportable agents has two weaknesses. There has been no formal characterization of the relationship between network, data and agent characteristics and agent performance. In addition current implementations focus on certain technical challenges to the exclusion of others. This thesis addresses these two weaknesses. Simulation and instrumentation will be used to explore the performance of transportable agents under different network conditions. In addition we will implement a complete transportable agent system that is flexible, secure and efficient. The implementation will involve identifying and extending existings solutions as well as developing novel ones. We have implemented a prototype system that uses Tcl as the agent language. This prototype shows promise and will be used as the starting point for the implementation work.

## 6 Acknowledgements

Many thanks to my advisor – Professor George Cybenko – for his encouragement and support; to the members of my thesis committee – Professor George Cybenko, Professor David Kotz, Professor Daniela Rus and Dr. Robert Sproull from Sun Microsystems – for their time and insight; to Keith Kotay for extensive discussion; to Yunxin Wu, Aditya Bhasin, Kurt Cohen and Katsuhiro Moizumi for implementing their information retrieval applications on top of the prototype system and providing useful feedback; and, as always, to Jennifer and Stephen Gray for reminding me that there is life outside graduate school.

## References

[BN84] A. D. Birrell and B. J. Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1):39–59, February 1984.

[BR93] Nathaniel Borenstein and Marshall T. Rose. Mime extensions for mail-enabled applications: application/Safe-Tcl and mulitpart/enabled-mail. Bellcore Memo, Bellcore, 1993. This memo is a working draft and should not be cited in published work.

[Car94] Luca Cardelli. Obliq: A language with distributed scope. Digital White Paper, Digital Equipment Corporation, Systems Research Center, 1994. This white paper is available at http://www.research.digital.com/SRC/Obliq/Obliq.html.

[Coe94] Michael D. Coen. SodaBot: A software agent environment and construction system. In Yannis Labrou and Tim Finin, editors, *Proceedings of the CIKM Workshop on Intelligent Information Agents, Third International Conference on Information and Knowledge Management (CIKM 94)*, Gaithersburg, Maryland, December 1994.

[Fal87] Joseph R. Falcone. A programmable interface language for heterogeneous systems. *ACM Transactions on Computer Systems*, 5(4):330–351, November 1987.

[Fon93] Leonard N. Foner. What's an agent anyway: A sociological case study. Agents Memo 93-01, Agents Group, MIT Media Lab, 1993.

[GG88] D. K. Gifford and N. Glasser. Remote pipes and procedures for efficient distributed communication. *ACM Transactions on Computer Systems*, 6(3):258–283, August 1988.

[GK94] Michael R. Genesereth and Steven P. Ketchpel. Software agents. *Communications of the ACM*, 37(7):48–53, July 1994.

[GSS94] Michael Genesereth, Narinder Singh, and Mustafa Syed. A distributed and anonymous knowledge sharing approach to software interoperation. In Yannis Labrou and Tim Finin, editors, *Proceedings of the CIKM Workshop on Intelligent Information Agents, Third International Conference on Information and Knowledge Management (CIKM 94)*, Gaithersburg, Maryland, December 1994.

[Haf95] Katie Hafner. Have your agent call my agent. *Newsweek*, 75(9), February 27 1995.

[jav94] The Java language: A white paper. Sun Microsystems White Paper, Sun Microsystems, 1994.

[Joh93] Raymond W. Johnson. Autonomous knowledge agents: How agents use the tool command language. In *Proceedings of the 1993 Tcl Workshop*, 1993.

[JvRS95] Dag Johansen, Robbert van Renesse, and Fred B. Scheidner. Operating system support for mobile agents. In *Proceedings of the 5th IEEE Workshop on Hot Topics in Operating Systems*, 1995.

[KK94] Keith Kotay and David Kotz. Transportable agents. In Yannis Labrou and Tim Finin, editors, *Proceedings of the CIKM Workshop on Intelligent Information Agents, Third International Conference on Information and Knowledge Management (CIKM 94)*, Gaithersburg, Maryland, December 1994.

[Mae94] Pattie Maes. Agents that reduce work and information overload. *Communications of the ACM*, 37(7):48–53, July 1994.

[Neg95] Nicholas Negroponte. *Being digital.* Alfred A. Knopf, 1995.

[OPL94] Tim Oates, M. V. Nagendra Prasad, and Victor Lesser. Networked information retrieval as distributed problem solving. In Yannis Labrou and Tim Finin, editors, *Proceedings of the CIKM Workshop on Intelligent Information Agents, Third International Conference on Information and Knowledge Management (CIKM 94)*, Gaithersburg, Maryland, December 1994.

[ora94] Oracle Mobile Agents. Oracle Press Release, Oracle, 1994.

[Ott94] Max Ott. Jodler - a scripting language for the infobahn. In *Proceedings of the 1994 Tcl Workshop*, 1994.

[Ous94] John K. Ousterhout. *Tcl and the Tk Toolkit.* Addison-Wesley Professional Computing Series. Addison-Wesley, Reading, Massachusetts, 1994.

[Ous95] John K. Ousterhout. Scripts and agents: The new software high ground. Invited Talk at 1995 Winter USENIX Conference, January 1995.

[Rei94] Andy Reinhardt. The network with smarts. *Byte*, pages 51–64, October 1994.

[Rie94] Doug Riecken. M: An architecture of distributed agents. *Communications of the ACM*, 37(7):48–53, July 1994.

[RN95] Stuart Russell and Peter Norvig. *Artificial intelligence: A modern approach*. Prentice-Hall Series on Artificial Intelligence. Prentice-Hall, Englewood Cliffs, New Jersey, 1995.

[Rog95] Adams Rogers. Is there a case for viruses? *Newsweek*, 75(9), February 27 1995.

[Sah94] Adam Sah. TC: An efficient implementation of the Tcl language. Master's thesis, University of California at Berkeley, May 1994. Available as technical report UCB-CSD-94-812.

[SBD94] Adam Sah, Jon Blow, and Brian Dennis. An introduction to the Rush language. In *Proceedings of the 1994 Tcl Workshop*, June 1994.

[SG90] J. Stamos and D. Gifford. Remote evaluation. *ACM Transactions on Programming Languages and Systems*, 12(4):537–565, October 1990.

[Spr95] Robert Sproull. Sun Microsystems, personal communication with the author, May 1995.

[SS94] Mukesh Singhal and Niranjan G. Shivaratri. *Advanced concepts in operating systems: Distributed, database and multiprocessor operating systems*. McGraw-Hill Series in Computer Science. McGraw-Hill, New York, 1994.

[Sto94] A. D. Stoyenko. SUPRA-RPC: SUbprogram PaRAmeters in Remote Procedure Calls. *Software-Practice and Experience*, 24(1):27–49, January 1994.

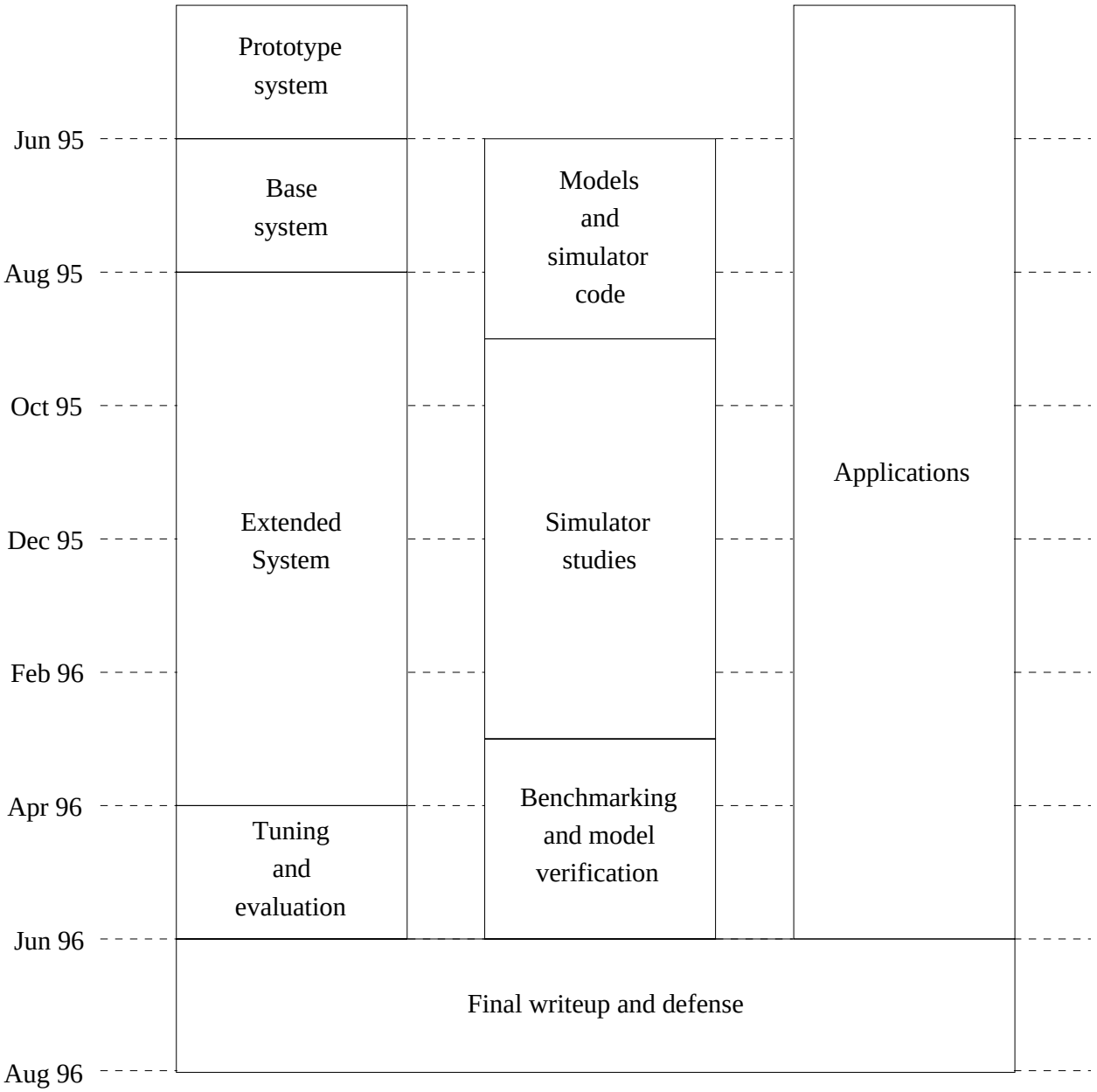
[Voo91] Ellen M. Voorhees. Using computerized routers to control product flow. In *Proceedings of the Twenty-Fourth Annual Hawaii International Conference on System Sciences*, pages 275–282. IEEE, January 1991.

[Whi94] James E. White. Telescript technology: The foundation for the electronic marketplace. General Magic White Paper, General Magic, 1994.

[WVF89] C. Daniel Wolfson, Ellen M. Voorhees, and Maura M. Flatley. Intelligent routers. In *Proceedings of the Ninth International Conference on Distributed Computing Systems*, pages 371–376. IEEE, June 1989.

# A Schedule

Below is the schedule for the thesis work. The prototype system has been implemented and is described in section 4. The base system adds synchronization, security, access to external resources and improved communication. The extended system adds the additional components discussed in section 3.




This is a reasonable schedule for three reasons - I am a fast, proficient programmer; I am finished with formal coursework; and other people have already begun to build applications on top of the prototype.

# B Documentation

This documentation was provided to the two groups that are using the prototype system. People who are not in these two groups will need to contact the author to obtain the source code for the prototype system or an account on one of the system on which the prototype is running get an account on one of the systems on which the prototype is running. The latter is limited to people that the author personally knows.

## B.1 Tcl

The transportable agents are written in the Tool Command Language (Tcl). Tcl was created by Dr. John Ousterhout at the University of Berkeley in 1987 and has enjoyed enormous popularity since then. Tcl is a general-purpose scripting language that has two components. The first component is a stand-alone shell similar to the C and Korn shells. The shell allows the user to interactively execute Tcl commands and scripts. The second component is a library of C functions. The library provides functions to create a Tcl interpreter, define new Tcl commands and submit Tcl scripts to the interpreter for evaluation. The library allows Tcl to be *embedded* inside any desired application – i.e. an application that needs a scripting language can include the library and have the user write Tcl scripts. The application can define application-specific commands for use in the scripts.

A tutorial on Tcl is beyond the scope of this documentation. However Tcl is relatively easy to learn and is similar to other scripting languages such as Perl and C shell. For example the following Tcl script computes factorials.

```
#!tclsh

proc factorial x {

    if {$x <= 1} {
        return 1;
    }

    expr $x * [factorial [expr $x - 1]]
}

set number 0

while {$number != -1} {

  puts "Enter a nonnegative integer (-1 to quit): "
  gets stdin number

  if {$number != -1} {
    puts "$number! is equal to [factorial $number]"
  }
}
```

The two most important aspects of this script are the *command* and *variable* subsitutions. For example in the command

```
expr $x * [factorial [expr $x - 1]]
```

$x is a variable substitution and [factorial [expr $x - 1]] is a command substitution. $x will be replaced by the value of variable x when the command is evaluated. [factorial [expr $x - 1]] will be replaced by the result of the command *factorial [expr $x - 1]*.

There are two ways to execute the script. If the Tcl shell is called *tclsh* and the script is in file *factorial*, then you can type *tclsh* to start the shell and then *source factorial* to load and execute the script. Alternatively you can turn on the execution permission for file *factorial* and just type *factorial*. This works because the first line in the script file is

```
#!tclsh
```

which tells the current shell – e.g. Korn shell or C shell – to run the script *using the tclsh shell*. In other words the current shell will start tclsh which will then run factorial. tclsh will terminate when factorial terminates.

There are numerous sources of information on Tcl. I recommend the Tcl book by Dr. Ousterhout [Ous94] and the Tcl news group *comp.lang.tcl*. I have two copies of the Tcl book and can help with whatever Tcl questions you have.

## B.2 Transportable agents

### B.2.1 Server

The transportable agent system is an extension of Tcl. A transportable agent is just a Tcl script that uses a special set of commands. The system has two components. The first component is a server that accepts Tcl scripts from remote machines and executes those scripts on the current machine. Transportable agents can be sent *only to those machines on which the server is running*. There are currently two sets of servers. Members of the artificial intelligence class can use the servers on

```
muir.cs.dartmouth.edu
tioga.cs.dartmouth.edu
tenaya.cs.dartmouth.edu
tuolomne.cs.dartmouth.edu
```

These machines are all DEC alphas. Members of the Thayer research group can use the servers on

```
bald.cs.dartmouth.edu
cosmo.dartmouth.edu
lost-ark.dartmouth.edu
temple-doom.dartmouth.edu
```

*bald* is an IBM-compatible personal computer running a shareware distribution of Linux; *cosmo* is an IBM workstation; *lost-ark* and *temple-doom* are Silicon Graphics workstations. Please be careful when using *bald* since it is exceptionally easy to overwhelm the networking component of the Linux kernel. The networking component will stop working if *bald* receives more than a few scripts or messages per second. The two sets of servers are completely disjoint. It is impossible to transport an agent from a server in one set to a server in the other. In addition the server can be installed on a new machine in less than ten minutes so please see me if you need the server to run on your "favorite" machine or on a larger number of machines.

### B.2.2 Agent shell

The second component of the system is an agent shell. This shell is the same as the Tcl shell *tclsh* except for the addition of agent commands and variables. The commands move a Tcl script from one machine to the other. The variables keep track of the current network location of the script. The shell is called *agent* and is found in directory

```
~bob/CODE/agent-tcl
```

| | |
|---|---|
| agent(local-server) | Name of the local server – i.e. the server that controls the agent |
| agent(local-id) | Numeric id that the local server has assigned to the agent |
| agent(root-server) | Name of the root server - i.e. the server that controls the *top-level* agent |
| agent(root-id) | Numeric id that the root server has assigned to the *top-level* agent |

Table 2: Agent variables

on cosmo and in directory

```
~rgray/CODE/agent-tcl
```

on all other machines. The process of running a script with the agent shell is the same as for the Tcl shell except that the first line of the script should be changed to

```
#!agent
```

and the shell is started by typing *agent* rather than *tclsh*. The agent shell can run on a machine even if the server is not running on that machine. Remember however that transportable agents can be *sent* only to those machines on which the server is running.

### B.2.3 Variables

The agent shell provides four variables that keep track of the network location of the agent. The variable *agent(local-server)* is the name of the machine whose server currently controls the agent. In general this is the name of the machine on which the agent is currently executing. Exceptions are discussed below. The server that currently controls the agent will be referred to as the *local server*. The variable *agent(local-id)* is the integer id that the local server has assigned to the agent.

Transportable agents can create child agents. This leads to a parent-child hierarchy with a single agent at the top. This agent is the *top-level* agent for itself and all of its children. The variable *agent(root-server)* is the name of the machine whose server controls the top-level agent. In general this is the name of the machine on which the top-level agent is executing. Exceptions are discussed below. The server that controls the top-level agent will be referred to as the *root server*. The variable *agent(root-id)* is the integer id that the root server has assigned to the top-level agent.

These variables are *global* variables and are always available inside an agent script. The variables are read-only. Their values change automatically as the agent moves through the network. Table 2 is a summary of the variables.

### B.2.4 Commands

The agent shell provides seven commands that are used to move a Tcl script through the network, send messages to another agent and receive messages.

1. agent_begin *server*

The top-level agent uses agent_begin to acquire a controlling server. The *server* argument is the name of the machine on which the server is running. For example if the top-level agent issues the command

```
agent_begin muir.cs.dartmouth.edu
```

then the server on *muir* becomes the controlling server for the agent. The top-level agent must always issue an agent_begin command since an agent can not create child agents or send and receive messages until it has a controlling server. In general an agent should use the server on its own machine as the controlling server.

This reduces the amount of network traffic. The most compelling reason to use the server on a different machine is if there is no server on the agent's machine.

The agent_begin command returns a two-element list on success. The first element of the list is the name of the controlling server. The second element of the list is the integer id that the server has assigned to the agent. In addition the command sets the four agent variables. agent(root-server) and agent(local-server) are set to the name of the server. agent(root-id) and agent(local-id) are set to the integer id. In other words the root and local id's of the top-level agent are the same. The children of the top-level agent will have the same root id but different local id's. Before the agent_begin command is issued, agent(root-server) and agent(local-server) are the empty string and agent(local-id) and agent(root-id) are -1.

agent_begin can fail. In this case the command raises a standard Tcl error and returns an error message. The possible error messages are

```
wrong number of arguments
agent has been registered
unable to send to server
server unable to comply (no response)
server unable to comply (server error)
server unable to comply (bad response)
```

The first message means that you provided the wrong number of arguments. The second message means that the agent already has a controlling server. The last four messages generally indicate a transient error due to network contention. Thus the appropriate action is to try the command again. If trying again does not work, please bring the problem to my attention immediately since the server has undoubtedly crashed. Server crashes and and network errors are rare. I have numerous agents that do no error checking at all and have never failed. However a robust agent should check for these errors. Since a Tcl error causes the script to halt – this is standard Tcl semantics – you will need to encapsulate the *agent_begin* command within a *catch* command in order to check for errors.

2. agent_end

The top-level agent uses agent_end to tell the controlling server that it no longer needs the server's services. The server removes the agent from its internal tables. In general agent_end should be called just before the top-level agent exits. On success agent_end returns an empty string. In addition it resets agent(root-server) and agent(local-server) to the empty string and agent(root-id) and agent(local-id) to -1. On failure agent_end raises a standard Tcl error and returns an error message. The possible error messages are

```
wrong number of arguments
agent has NOT been registered
unable to send to server
server unable to comply (no response)
server unable to comply (server error)
server unable to comply (bad response)
```

The first message means that you have provided the wrong number of arguments. The second message means that the agent does not have a controlling server. The last four messages are the same as for agent_begin. They indicate a server crash or a transient error due to network contention.

In light of the previous discussion all transportable agents have the form

```
#!agent
# This is the top-level agent.

# preprocessing
```

```
agent_begin muir        # or any other machine with a server

# processing
# create child agents
# send and receive messages

agent_end

# postprocessing
```

3. agent_send *destination* [*integer*] *string*

agent_send sends a message to another agent. A message consists of an integer code and a string. *integer* is the integer code. The code defaults to 0 if it is not specified. *string* is the string. *destination* specifies the destination agent. *destination* is a two-element list where the the first element is the name of the destination agent's local server and the second element is the destination agent's local integer id. For example the following command sends a message to the agent with local integer id 4 on *tuolomne*.

```
agent_send "tuolomne 4" 2 "DOC 1 = YES, DOC 2 = NO, DOC 3 = YES, DOC 4 = NO"
```

agent_send returns the empty string on success. agent_send raises a standard Tcl error and returns an error message on failure. The possible error messages are

```
wrong number of arguments
agent has NOT been registered
second argument must be a two-element list: server followed by id
the id must be an integer
unable to send to server
server unable to comply (no response)
server unable to comply (server error)
server unable to comply (bad response)
```

The first four error messages are self-explanatory or the same as before. The last four error messages are the same as before except that *server unable to comply (server error)* probably means that the specified agent *does not exist*. However it could indicate a server crash or a transient network error as before. These two cases will be split into separate error messages soon. You must be careful when using agent_send. If the destination agent moves to a new machine, it is assigned a new local server and a new local integer id. You must use the new id when sending messages. Any message sent to the old id will be lost. The system does not have the capability to forward messages to the new location. This is a weakness of the current implementation.

4. agent_receive *variable*

agent_receive is used to receive a message that has been sent to the agent. If no message is available, agent_receive returns -1 and sets the variable to the empty string. If a message is available, agent_receives returns the message code and sets the variable to the message string. agent_receive is a nonblocking call. It returns -1 immediately if no message is available. Therefore most calls to agent_receive are placed inside a loop that iterates until agent_receive returns a code other than -1. The lack of a blocking capability is a weakness of the current implementation. Continuing with the agent_send example from above, suppose that the agent with local integer id 4 on *tuolomne* issues the command

```
agent\_receive val
```

The command returns 2 since this is the message code that was specified in the agent_send. In addition the command sets val to "DOC 1 = YES, DOC 2 = NO, DOC 3 = YES, DOC 4 = NO" since this is the

message string that was specified in the agent_send. If for some reason the message had not arrived when agent_receive command was issued, agent_receive would return -1 and the script would have to issue the command again.

agent_receive can fail with one of the following error messages.

```
wrong number of arguments
agent has NOT been registered
unable to send to server
server unable to comply (no response)
server unable to comply (server error)
server unable to comply (bad response)
unable to set variable
```

These errors are the same as before except for *unable to set variable*. This means that the agent_receive command was unable to access and set the specified variable. Either the variable is read-only or a pathological error has occurred. Running out of memory is an example of a pathological error. Pathological errors are rare in correct Tcl scripts.

5. agent_submit *server* [-procs *name name ...*] [-vars *name name ...*] -script *script*

agent_submit submits the Tcl script *script* to server *server*. The script becomes an agent under that server's control. The agent that issues the agent_submit command is the *parent* of the new agent. If the new agent needs to use variables or procedures that have been defined in the parent, the names of these variables and procedures are specified in the agent_submit command after the *-vars* and *-procs* flags. The variables and procedures are sent to the server along with the script. Note that the transmitted variables are *copies*. There is no connection between the child and parent variables. Changes in one are never seen in the other. In addition the transmitted variables become *global* variables in the child.

The agent_submit command returns a two-element list on success. The first element is the name of the server to which the child was submitted. The second element is the integer id that the server has assigned to the child. In other words the command returns the local server and local integer id of the new child. In the child the variables agent(root-server) and agent(root-id) are the same as in the parent. The variables agent(local-server) and agent(local-id) are set to the local server and local integer id of the child.

Every Tcl script has a result. The result of the script is the result of the last command executed. When the child agent finishes executing and terminates, a message containing the script result is *automatically* sent to the top-level agent – i.e. the agent specified in agent(root-server) and agent(root-id). The top-level agent uses agent_receive to receive the result. The message string is the script result. The message code indicates the way in which the child terminated. The four possible message codes are

```
0 = normal
1 = error
3 = break command was issued
4 = continue command was issued
```

As an example the following agent asks for an integer and then computes the factorial of the integer on a *different machine*. For clarity the script does not perform any error checking.

```
#!/a/quimby/usr/toe/grad/rgray/CODE/agent-tcl/agent

proc factorial x {

    if {$x <= 1} {
        return 1;
    }
```

```
    expr $x * [factorial [expr $x - 1]]
}

set number 0

  # make muir the controlling server

agent_begin muir

while {$number != -1} {

  puts "Enter a nonnegative integer (-1 to quit): "
  gets stdin number

  if {$number != -1} {

       # compute the factorial on tioga.  We submit the script "factorial
       # $number" along with procedure "factorial" and variable "number"
       # since the script needs these two things.

    agent_submit tioga -vars number -procs factorial \
                       -script {factorial $number}


       # the result of the submitted agent is the result of the last command
       # executed -- i.e. the return value of the factorial procedure.
       # When the submitted agent ends, this result is automatically sent to
       # the top-level agent (which is this agent).   We just loop until we
       # receive the message.

    set result -1

    while {$result == -1} {
      set result [agent_receive value]
    }

    puts "$number! is equal to $value"
  }
}

  # tell muir that the agent is done

agent_end
```

agent_submit can fail with one of the following error messages.

```
wrong number of arguments
you must specify a script
"-script" must be followed by a script
procedure XXX does not exist
variable XXX does not exit
unable to send to server
server unable to comply (no response)
```

```
server unable to comply (server error)
server unable to comply (bad response)
```

These error messages are self explanatory or the same as before.

6. agent_jump *server*

The agent_jump command suspends the execution of the agent. The agent is transported to server *server* where it resumes execution *at the point at which it was suspended.* agent_jump returns the empty string on success. In addition agent(local-server) and agent(local-id) are set to the new local server and the new local integer id. agent(root-server) and agent(root-id) are unchanged. The *top-level* agent can not jump. Since there is no message forwarding capability, if the *top-level* agent jumps to a new location, all of the results that are automatically sent back to the top-level agent will be lost. In addition the agent will lose its connection to the display device and can not regain the connection even when it jumps back to the home machine. If you want an agent that jumps, you should have the top-level agent submit a second agent using agent_submit. The second agent can then jump at will. As an example here are two agents. The first agent executes the *who* command on multiple machines by submitting one agent to each machine. Note that the agent performs no error checking.

```
#!/a/quimby/usr/toe/grad/rgray/CODE/agent-tcl/agent

set num 4
set machine1 {muir.cs.dartmouth.edu}
set machine2 {tenaya.cs.dartmouth.edu}
set machine3 {tioga.cs.dartmouth.edu}
set machine4 {tuolomne.cs.dartmouth.edu}

  # start up the agent

agent_begin tioga.cs.dartmouth.edu

  # submit child agents on each machine
  # each child agent does an "exec who" to get the list of logged-on users

for {set i 1} {$i <= $num} {incr i} {
  agent_submit [set machine$i] -script {
    set who_info [exec who]
    set location $agent(local-server)
    set result "$location:\n$who_info"
  }
}

  # collect the results from each child agent
  # these results are automatically sent by the children

set r_count 0;

while {$r_count < $num} {
  if {[agent_receive result] != -1} {
    incr r_count;
    puts $result
  }
}

agent_end
```

The second agent executes the *who* command on multiple machines by having a single agent jump from machine to machine.

```
#!/a/quimby/usr/toe/grad/rgray/CODE/agent-tcl/agent

set num 4
set machine(1) {muir.cs.dartmouth.edu}
set machine(2) {tenaya.cs.dartmouth.edu}
set machine(3) {tioga.cs.dartmouth.edu}
set machine(4) {tuolomne.cs.dartmouth.edu}

agent_begin tioga

  # submit the script that will jump from machine to machine.  We send the
  # scalar variable "num" and the array "machine" since the script needs
  # these two variables. Note the msg variable in the submitted script.  At
  # each machine we append the who list for that machine to msg.

agent_submit tioga -vars num machine  \
    -script {

       set msg ""

       for {set i 1} {$i <= $num} {incr i} {
         agent_jump [set machine($i)]
         set who_info [exec who]
         set location $agent(local-server)
         append msg "$location:\n$who_info\n"
       }

         # we want to return the concatenated who lists so we just do
         # a return $msg -- this is the last command in the submitted
         # script so the script result ends up being the value of msg

       return $msg
}

  # get the result from the submitted script
  # this result is automatically returned

set result -1

while {$result == -1} {
  set result [agent_receive msg]
}

puts $msg
agent_end
```

Both of these scripts produced the following output during a test run.

```
tioga.cs.dartmouth.edu:
rgray       ttyp1       Mar 29 13:49
```

```
songbac     ttyp2       Mar 29 10:53
tuolomne.cs.dartmouth.edu:
rgray
tenaya.cs.dartmouth.edu:
rgray
muir.cs.dartmouth.edu:
rgray
```

agent_fork can fail with one of the following error messages.

```
wrong number of arguments
unable to send to server
server unable to comply (no response)
server unable to comply (server error)
server unable to comply (bad response)
```

These error message are the same as before. Note that the agent *never* changes location when agent_jump fails.

7. agent_fork *server*

agent_fork is analogous to Unix fork. The command creates a copy of the agent on the specified server. The parent and the child then continue execution from the point at which the fork occurred. On success agent_fork will return a two-element list in the parent and the string "CHILD" in the child. The two-element list contains the local server and local integer id of the child agent. In the child agent(local-server) and agent(local-id) are set to the local server and the local integer id. agent(root-server) and agent(root-id) are the same as in the parent. All variables are unchanged in the parent. As an example here is the skeleton of a top-level agent that performs a fork.

```
#!/a/quimby/usr/toe/grad/rgray/CODE/agent-tcl/agent

agent_begin muir

if {[agent_fork tioga] == "CHILD"} {

  # child processing here

} else {

  # parent processing here
  # wherever you put the agent_end, make sure that only the parent does it

  agent_end

}
```

agent_fork can fail with one of the following error messages.

```
wrong number of arguments
unable to send to server
server unable to comply (no response)
server unable to comply (server error)
server unable to comply (bad response)
```

These error message are the same as before. Note that a child is *never* created when agent_fork fails.

Table 3 summaries the agent commands. The scripts that were developed above are in

`~bob/CODE/agent-tcl/scripts`

on *cosmo* and

`~rgray/CODE/agent-tcl/scripts`

on all other machines. The best way to get a feel for the system is to first understand the sample scripts. Then start up the agent shell and try some of the commands *interactively*. Note that the agent_send command is hard to try interactively. Then write some *simple* agents.

### B.2.5 Caveats

There are several limitations with the current system asides from the ones metioned above. First there are four commands that the standard Tcl shell supports but the agent shell does not.

1. All *history list* commands are not available in the agent shell.
2. The *case* command is not available.
3. The *info script* command is not available.
4. The *info level* command is available but you can not use the form that takes an argument – i.e. *info level integer* is not available.

These commands – except for the *case* command which is anachronistic – will be supported soon. However please do not develop a transportable agent that requires them since I can not guarantee any particular time frame.

Second there are some limitations on agent_fork and agent_submit. These two commands capture the internal state of an executing Tcl script and transmit this state to another machine. The limitations relate to those portions of the state that currently can not be captured.

1. Do not *unset* an array element that has an *upvar* pointing to it and then jump or fork. The array element will not be captured in the state image and will not be available on the destination machine. There is rarely a need to unset such an array element so this is not a a large problem.
2. Do not fork or jump inside a variable trace or inside a procedure that the sort command is using to compare list elements. The state image will not be captured correctly. This should never be done anyways since it is a terrible side effect – e.g. the script has moved to a new machine in the middle of the sort.
3. The following portions of the state are ignored: deletion callbacks, open files, linked variables, variable traces, command traces, interrupt handlers, child processes, array searches, user-defined math functions and the internal state of all *Tcl extensions*. This is not a large problem since most of these are inherently nontransportable due to their close ties to a particular machine or to underlying C code. The rest such as array searches, user-defined math functions and the internal state of Tcl extensions do not need to be transmitted often. Thus this should not be a limiting factor in initial agent development. Note that you can use all of the listed constructs. The limitation is that you can not create or define the construct on one machine and then transmit it to another machine.

All of these constructs are described in [Ous94]. The most important thing to note is that it is nearly impossible to use them accidentally. If you do not know what they are *or* do not think that you are using them, you are not using them.

| | |
|---|---|
| agent_begin *server* | Tell the server that an agent has started |
| agent_end | Tell the server that the agent has finished |
| agent_submit *server*<br>[-procs *name name ...*]<br>[-vars *name name ...*]<br>-script *script* | Execute a script on a remote machine |
| agent_fork *server* | Create a copy of the agent on a remote machine |
| agent_jump *server* | Transfer the agent to a remote machine |
| agent_receive *variable* | Receive a message |
| agent_send *id* [*integer*] *string* | Send a message |

Table 3: Agent commands

### B.2.6 Security

The system has no security features and does not identify the user who has submitted the agent. I have not reached the security part of the implementation. Currently all submitted agents run with *my userid* and *my access permissions*. This means two things. (1) Please be careful with your agents so that you do not delete my files or the files of someone using the agent system. This is not a large concern since you can not possibly affect a file unless your agent explicitly issues a remove command or explicitly opens the file. (2) Your script can do only what *my userid* has the authority to do. You will have to make sure that my userid has the necessary access rights. In general this should mean nothing more than making some files publicly readable or a particular directory publicly writable.

### B.2.7 Advanced

There is a library of C functions associated with the agent shell that is similar to the Tcl library. This allows transportable agent capability to be embedded in other applications. In addition all extensions that can be used with standard Tcl can be used with the agent version. The compiled agent library is linked with the compiled extension library as usual. If you want to embed the system in another application or use one of the extensions – such as Tk which provides an interface between Tcl and X-windows – I can provide necessary guidance.