DADS - A DISTRIBUTED AGENT DELIVERY SYSTEM

Clifford Joseph Cozzolino, B.S.

Thesis Prepared for the Degree of

MASTER OF SCIENCE

UNIVERSITY OF NORTH TEXAS

December 2002

APPROVED:

Armin R. Mikler, Major Professor
Azzedine Boukerche, Committee Member
Steve Tate, Committee Member
Paul Tarau, Committee Member
Robert Brazile, Graduate Advisor
Krishna Kavi, Chair of the Department of Computer Science
C. Neal Tate, Dean of the Robert B. Toulouse School of Graduate Studies

Cozzolino, Clifford Joseph, DADS - A Distributed Agent Delivery System, Master of Science (Computer Science), December 2002, 98 pp., 2 tables, 19 figures, 27 titles.

Mobile agents require an appropriate platform that can facilitate their migration and execution. In particular, the design and implementation of such a system must balance several factors that will ensure that its constituent agents are executed without problems. Besides the basic requirements of migration and execution, an agent system must also provide mechanisms to ensure the security and survivability of an agent when it migrates between hosts. In addition, the system should be simple enough to facilitate its widespread use across large scale networks (i.e Internet).

To address these issues, this thesis discusses the design and implementation of the Distributed Agent Delivery System (DADS). The DADS provides a de-coupled design that separates agent acceptance from agent execution. Using functional modules, the DADS provides services ranging from language execution and security to fault-tolerance and compression. Modules allow the administrator(s) of hosts to declare, at run-time, the services that they want to provide. Since each administrative domain is different, the DADS provides a platform that can be adapted to exchange heterogeneous blends of agents across large scale networks.

ACKNOWLEDGMENTS

I wish to express my utmost gratitude and appreciation to Dr. Armin Mikler. His infinite patience and unabated perseverance has motivated me to achieve excellence in all aspects of my work. His guidance, insightful conversation, and approach to new ideas have not only helped me gain insight into my own research, but they have also helped me build a foundation of confidence in myself for which I am forever grateful.

I would like to thank my committee members, namely Dr. Steve Tate, Dr. Azzedine Boukerche, and Dr. Paul Tarau for participating in both the review and defense of this thesis. Not only do I greatly appreciate their comments and suggestions, but their instruction and guidance throughout my undergraduate and graduate career have been invaluable.

Many thanks go to Shubashini Raghunathan and John Mayes for their invaluable suggestions, comments, and advice regarding the design and implementation of the DADS system. Also, I'd like to thank all of the members of the NRL, especially the Agent Research Group (ARG), for listening to my ideas and offering suggestions to help improve my work.

Also, I'd like to thank all of my friends and family for their moral support throughout my academic career. Their support has given me the strength to continue to work, even when it seemed that the odds were against me.

Finally, and most importantly, I would like to thank my parents. Their unfaltering encouragement has given me the courage and desire that has allowed me to strive toward and achieve my goals. Without their infinite love and support, none of this would have ever been possible.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

The concept of an intelligent agent has been around for quite some time. First described in the literature of artificial intelligence, an agent is code that acts on behalf of a user [20]. In addition to code, an agent carries some associated data. This facilitates autonomy, where an agent can perform tasks with little to no user interaction. Since they are programs, agents require a platform that will execute their instructions. In the context of an operating system, an agent may execute as a running process or as code executing in an interpreter. For an interpreted agent, code executes within the context of an interpreter and uses the functionality provided by it. Regardless of how an agent is executed, the underlying platform that makes agent execution possible is defined as the agent's *infrastructure*. Unless an infrastructure provides conventions for code mobility, agents remain on the local host, making them static. Therefore, before an infrastructure can support code mobility, it must first provide functionality that will facilitate the transfer of an agent's code and data to other hosts.

An infrastructure facilitates the execution of the instructions that are carried by

an agent. In addition, it makes resources available to the agents that use it. Given the intimate relationship between an agent's code and execution platform, many infrastructures employ a *virtual machine* (e.g., Java[1] Virtual Machine) where agent functionality is incorporated into the execution platform directly, making it *agent-oriented*. In this context, agents are written in a high level language and compiled into a byte-code that is specific to the virtual machine. This facilitates portability since the virtual machine provides an execution platform that is independent of a host's architecture. However, the virtual machine model has its disadvantages. For instance, virtual machines place a number of limitations on the agents that it can support. Since it must use the byte-code dictated by the particular virtual machine, an agent is subject to the fundamental principles of the virtual machine core design (i.e., security, performance). This may lead to a design that reduces the overall effectiveness of the agent paradigm since it creates an environment that may be geared toward certain applications. Also, since there is no universal virtual machine that recognizes all languages [26], the choice of language is limited when developers need to select a language for agent development.

Our goal is to move agents away from the limitations of a single virtual machine architecture, making the availability of the agent paradigm more widespread. Similar

[1]® Sun Microsystems Inc., Java, December 7 1999, http://java.sun.com

to standard network services such as Telnet or the File Transfer Protocol (FTP), hosts within a network could offer a standard service designed specifically for agents. When a host executes an FTP daemon, that host makes file transfer service available for all to use. Similarly, we would like to see the agent paradigm follow this same principle. Instead of an infrastructure that supports one specific type of an agent, we would like to see a system that can accept and execute a heterogeneous blend of agents. Since it is not designed around the specifications of one particular virtual machine, such a platform can encompass a larger set of hosts, thereby providing access to an extended number of computing resources. However, increased access to resources introduces several new issues that must be considered. First, a security model must be developed that is flexible enough to secure both a host and agent. Since heterogeneous agents may be accepted, different agents will inevitably use different forms of security. Second, to facilitate heterogeneity, an agent infrastructure must be multi-lingual. That is, compilers and interpreters for different languages must be made available at the constituent sites. And third, the agent infrastructure must support robust mechanisms for fault tolerance. Since an agent executes autonomously, it is the responsibility of the underlying agent platform to provide methods for recovery in the event of a fatal error. In general, different hosts may be configured with dissimilar services, thus, it is imperative that the availability of a particular service be advertised

to incoming agents. This provides an agent with sufficient information to determine whether the host meets its resource requirements. To facilitate the communication of this information, a protocol must be developed.

In order to build an agent platform it is necessary to review the requirements of an agent and the environment in which it executes. With important issues such as performance, security, and fault tolerance, an infrastructure must balance several factors to obtain efficient and secure agent delivery across heterogeneous networks. Unfortunately, diversity among existing agent systems has made the adoption of an agent standard difficult. As a result, many infrastructures have used agents and protocols of their own design, thereby reducing some of the potential for interaction between infrastructures. To address some of these issues, we introduce the Distributed Agent Delivery System (DADS).

Executed as a daemon process that listens for incoming agents on a network, the DADS acts as a portal to computing resources. Using a plug-in style interface, these resources are made available as loadable services, referred to as *modules*. Influenced by the organization of AgentTcl [12], a module can be used to provide language services. In addition, DADS modules also provide services such as compilation, security, fault tolerance, and data compression. In general, modules facilitate the dynamic loading of services, hence, they can be brought online when they are needed and taken off-line

when they are idle. When compared to systems that define their services at build time, a dynamic configuration is much more attractive.

Influenced by the agent design introduced in the Tromsø And COrnell Moving Agents (TACOMA) system [19], a DADS agent consists of three segments: code, data, and properties. Using the terminology of TACOMA, a DADS agent is comparable to a briefcase containing three folders. The code segment stores the instructions that are executed by an agent. The contents of the data segment represent the agent's state. And the properties segment stores a description of both the code and data segments. Combined, these three segments make up the network transmittable structure referred to as a DADS agent.

The code and data segments are designed to store arbitrary sequences of bytes, hence, a DADS agent is inherently multi-lingual. As an autonomous entity, an agent must be able interact with other hosts. Since autonomy dictates no user interaction, it is imperative that the agent maintain a description of itself (i.e., language, security, etc.) to communicate. Without a description, agents would not be able to decide which hosts can facilitate execution and guarantee the agent's survival. To address this, a DADS agent uses a *properties* segment to describe its contents. Influenced by the concept of a *badge* in Mole [4], the properties segment is designed to facilitate agent migration and agent-to-agent interaction. That is, it provides an agent with a

description of itself (i.e., language, security, etc.) that can be used during the transfer protocol or when agents meet. As work on an agent standard continues[10, 7], the properties segment can be easily adapted to conform to a standards based description.

Related Work

In recent years, many discoveries have been made in agent based computing. Most important have been the advancements made in the area of agent systems. Building on some of the basic principles discussed above, research in agent systems has introduced many new and interesting concepts to the area of code mobility. In addition, ongoing research has promoted the exchange of ideas, creating a new generation of hybrid systems. Subscribing to a hybrid model, the DADS expands upon some of the concepts found in some of today's systems. In this section, we introduce some of the systems that have influenced the DADS design. In addition, we briefly describe some of today's proposed agent standards.

Telescript Designed primarily for a proprietary hand-held device, Telescript[2] [27] is one of the first mobile agent systems ever developed. A pioneer in the field, Telescript is one of the first systems to support agents using a virtual machine architecture. A predecessor to the Java Virtual Machine (JVM), Telescript uses a *Telescript Engine*

[2]® General Magic Inc., Telescript, June 4 1996, http://www.genmagic.com

to execute byte-code instructions that are compiled from the Telescript language. Similar to Java, Telescript uses an object model to facilitate the implementation of what is known as its *agents* and *places*. Comparable to the initial agent definition given above, an agent is an entity that encompasses the code mobility aspect of the Telescript system. Utilizing a secure virtual machine architecture, Telescript restricts its agents from directly accessing the underlying operating system on a host. Instead, it makes the agents interact with what is referred to as a place. In contrast to an agent, a place is delegated access to certain resources on a host. Coincidentally, a place is simply a static agent that is used as a proxy to resources. A host may offer multiple places, hence, each place may offer a different service. Further, multiple agents can interact with any number of places concurrently. Thus, whenever a Telescript agent migrates, the agent does not access resources on the host directly. Instead, an agent interacts with other agents and places via *meetings*, which facilitate the setup and execution of inter-agent communication and information exchange. Best described as an *electronic marketplace* [27], the principles of agents and places is analogous to the interactions of humans in the physical world.

Telescript has introduced several concepts to the mobile agent paradigm. In addition to places and meetings, Telescript is the first agent system to introduce the concept of single-instruction agent migration. That is, an agent can migrate with a

simple call to the *go* function. This suspends agent execution, packages the agent, and transmits it to a destination. This avoids having an agent provide its own transfer functionality, thereby reducing its overall complexity.

Aglets An Application Programming Interface (API) for Java, Aglets[3] [18, 1] offer an agent infrastructure designed around the Java object model. Similar to Telescript, Aglets also use a virtual machine architecture, the JVM, to execute mobile objects. Developed under the name of an *Aglet*, these mobile objects execute as Java threads. In the Aglet system, a running Aglet (thread) is referred to as a *context*, where it is subject to all of the advantages (i.e., security, portability) offered by the JVM. Built upon the Java object model, Aglets are fairly easy to implement since many developers are already familiar with Java.

Using the same principles as the *go* instruction available in Telescript, Aglets provide mobility through a *dispatch* function. This function suspends a context, packages it, and transmits it to a destination host. At the destination, an object called a *listener* waits for an incoming agent transmission. Nothing more than specialized Aglets, listeners read the incoming contexts from the network, unpackages them, and resumes their execution on the host.

[3]® Internation Business Machines Corporation (IBM), Aglets, June 29 1999, http://www.trl.ibm.com/aglets/index_e.htm

The popularity, simplicity, and security of Java has made the Aglet infrastructure a popular choice within the agent research community. However, as mentioned by Jim White, one of the lead developers of Telescript:

> Telescript and Java virtual machines share one important trait; they institutionalize a particular object model...A better approach is a virtual machine that is language neutral. [26]

A disadvantage of the virtual machine, predefined object models require that all agent execution ultimately use the byte-code of that particular virtual machine. Since each agent system is designed to function with its own agents, predefined object models enforce a structured agent design. In addition, it reduces the interoperability of the agent with other systems since the byte-code cannot execute without the particular virtual machine. Also, developers are limited in their choice of agent language, hence, if the developer needs to use certain functionality of a language and it does not compile down to JVM byte-code, the developer might not be able to use the Aglet system at all.

AgentTcl One of the most flexible of today's mobile agent infrastructures, AgentTcl [12] (currently D'Agents) introduces several refreshing concepts to the area of agent mobility. Moving away from the single virtual machine design, AgentTcl employs

a collection of loadable interpreters for agent execution. Since different interpreters are made available concurrently, AgentTcl can handle multi-lingual agents. Further, since certain languages may be specialized for certain applications (i.e., speed, security, portability). The choice of language gives developers the flexibility to use an agent that fits the needs of their applications. In addition, a developer may choose a familiar language, thereby decreasing development time.

The AgentTcl infrastructure is divided into two halves. The first half consists of a single daemon process referred to as *agentd*. This daemon listens on the network for agents that wish to migrate. When a migration request is received, *agentd* accepts the transmitted agent. The second half, a collection of interpreters, is responsible for agent execution. Thus, when *agentd* receives an agent from the network, it forwards the agent to a loaded interpreter for execution.

During its execution, an interpreter maintains a context (i.e., code, stack, variables) that correlate to a program's execution. Also known as *state*, this information contains all of the data needed to execute the program within the particular interpreter. In AgentTcl, this state information constitutes an agent. That is, if we were to take a snapshot of a running program in an interpreter, the suspended context contains program code and data, which follows our definition for an agent. To facilitate mobility, AgentTcl uses a method known as *state-capture*. Using a modified

interpreter, agents can perform state-capture by making a function call similar to the *go* and *dispatch* functions used in Telescript and Aglets, respectively. In general, this call suspends execution, takes a snapshot of the current context, opens a connection to a remote AgentTcl host, and transmits it. At the remote host, this information is read and used to create a new context for continued execution.

The modular design offered by AgentTcl makes it a very attractive infrastructure for experimental agent research. Supporting languages such as Tcl, Python, Scheme, and Lisp, AgentTcl provides a flexible platform for multi-lingual agents. State-capture allows AgentTcl to be as efficient as possible by reducing the amount of time consumed during agent transfer, however, it also requires the addition of special functionality to an interpreter. That is, an interpreter must be modified in order to support the state-capture routines. Since there are many interpreted languages that do not support this functionality, users must wait for the AgentTcl development team to provide support. While it is possible for an AgentTcl user to add this support themselves, modification of an interpreter requires time and resources. Further, not all interpreters are open source, hence, code modification may not be possible without direct vendor involvement.

Mole Built upon the JVM, Mole [4] also uses a virtual machine architecture. Similar to Aglets, a Mole agent is simply a Java thread. In addition, Mole incorporates some of the features introduced by Telescript, namely the concept of an *agent* and a *place*. Similar to Telescript, agents are restricted direct access to a host's resources. Instead, Mole also uses a collection of places (i.e., static agents) to act as proxies to resources. Similar to a meeting, Mole agents initiate *sessions* to interact with places. Sessions facilitate the exchange of information between an agent and place.

Unique identification of an agent is important. Since many agents may exist concurrently across a large domain, it is advantageous to be able to identify one agent from another. Further, it would be advantageous for an agent to engage in sessions with agents that are performing similar tasks. Before a session can be established, an agent must know what service is provided by a particular place. Likewise, a unique identifier allows every agent and place to be accounted for. To address this, Mole uses a unique identification scheme. Designed to facilitate session setup, Mole's identification scheme combines the concept of a globally unique agent-id with an application specific identifier, known as a *badge*. Geared specifically for its mobile agents, a badge allows an agent to advertise its application specific properties. Thus, when an agent wants to engage in a session with another agent or place, the parties involved can use the badge information to determine which agents are performing

similar tasks.

TACOMA One of the earliest implementations of a mobile agent infrastructure, TACOMA [19] introduces some very interesting concepts to agent mobility. In its earliest version, TACOMA defines its agent as a Tcl procedure. Unlike systems that use virtual machine threads or captured state information, a TACOMA agent uses the code from a high level programming language. This means that state information is not implicitly coupled with the code as it is with a state-captured image. Instead, this information must be explicitly programmed and stored with the agent. To facilitate this, TACOMA employs a system of *briefcases* and *folders*. Forming a two level hierarchy, a briefcase may contain several folders, whereas a folder contains agent data. A folder may contain any arbitrary sequence of bits, thereby avoiding any limitations on the data that it contains. Further, a folder is referenced through a descriptive name. For instance, an agent may carry a briefcase that contains a folder labelled *CODE*. This folder may include the Tcl procedure defining the actions of that particular agent. Likewise, before an agent migrates, it must reference the folder labelled *HOST* to determine its destination.

Parallel to other infrastructures, TACOMA agents require interaction with other

agents for execution. Using a *meet* operation, a TACOMA agent exchanges its briefcase with another agent. The receiver opens the briefcase, references the *CODE* folder, and executes the associated code. The simplicity of this model is very intriguing. Since the contents of a folder are not restricted, an agent's contents are no longer bound to the specifications of the underlying infrastructure (i.e., language, architecture).

FIPA As more agent systems are developed, system interoperability becomes a problem. Since a host may execute more than one infrastructure concurrently, it would be advantageous for the agents of the concurrent infrastructures to communicate. Unfortunately, agents of one infrastructure may not understand the context in which other agents execute. To interact, one of the systems must be modified. Inevitably, this leads to the question of which system to modify. Since some systems are not open-source, modification of a system to fit the needs of another may be impossible. Also, many agent systems exist, hence, it would also be impractical to support the requirements of each. Consequently, a standard for interoperability has become imperative.

Designed to promote an open standard for agent system design, the FIPA specification is one of the most comprehensive specifications for agent systems thus far.

Described best by its mission statement, FIPA's mission is:

> The promotion of technologies and interoperability specifications that facilitate the end-to-end interworking of intelligent agent systems in modern commercial and industrial settings. [10]

FIPA provides an extensive collection of specifications designed to aid developers in building inter-operable infrastructures. Unfortunately, many of today's infrastructures do not support the FIPA standard, or for that matter, any standard at all. This not only reduces the effectiveness of a particular system, but it also reduces the effectiveness of the agent paradigm itself.

MASIF Developed by the Object Management Group (OMG[4]) [21], MASIF [7] provides another standard for interoperability among agent systems. Intended as an open standard, MASIF is built on top of the CORBA[5] system. Best stated by its specification, MASIF is *"a collection of definitions and interfaces that provide an interoperable interface for mobile agent systems"* [7]. MASIF does this by offering a standard definition for agent naming, management, and transfer which could be used across agent systems to accommodate interoperability.

[4]® Object Management Group Inc., OMG, September 29 1992, http://www.omg.org
[5]® Object Management Group Inc. CORBA, April 7 1998, http://www.omg.org

Grasshopper Grasshopper [5] is the first MASIF and FIPA compliant agent system ever developed. As discussed by the developers, *"Grasshopper is in principle a MASIF conformant mobile agent platform, which has been enhanced recently with a FIPA add on, in order to give the application developer total flexibility"* [5]. Written in Java, the Grasshopper platform is designed to provide a completely distributed agent environment capable of handling agents of varying complexities. Using the proposed definitions found in MASIF and FIPA (i.e., agent naming, management, transfer, etc.), Grasshopper offers a comprehensive, standards based solution to agent systems. Hence, it has been accepted for use in many applications including telecommunications and e-commerce.

DADS

Influenced heavily by the design and implementation of the systems above, the DADS provides an agent platform for the acceptance and execution of heterogeneous agents. Using a modular design, the DADS provides a virtual gateway to services. Similar to the concept of a place, DADS services are realized in the form of modules, where a module provides any number of services ranging from language execution and security to fault-tolerance and compression. Modules separate agent services from the core delivery functionality, thereby offering a platform that can support a wider range

of agents. Coupled with an agent structure that acts as a mobile container for code and data, the DADS is an agent system focused on flexibility. As a result, the DADS uses a unique agent delivery protocol to facilitate heterogeneous agent delivery across multiple domains (i.e., corporations, universities). Further, when it is needed, this flexibility allows the DADS to support the proposed standards discussed above.

In what follows, we present the design and implementation of the DADS system. Beginning with Chapter 2, we discuss some of the pertinent issues encountered during the design of the DADS. Further, it presents a brief overview of the DADS organization, as well as a discussion of how today's agent systems have influenced the DADS design.

Following this discussion, the DADS agent is introduced. Specifically, we discuss its structure, which leads into a description of the *property structure* used throughout the DADS system. Further, a development strategy for the DADS agent is presented.

Next, Chapter 2 introduces the DADS daemon. In particular, we discuss the requirements of a host that would like to participate in an agent system, focusing primarily on how it must facilitate agent migration and execution. Next, a development strategy for the DADS daemon is presented.

The DADS uses a modular design to provide services. Managed directly by the DADS daemon, modules are an important feature of the DADS system. Consequently,

we also discuss the module system, focusing on its design and use. Further, we supplement this discussion with a concept referred to as *module chaining*, and follow up with suggestions for future module development. Finally, a development strategy for a basic module is presented.

Chapter 2 concludes with a discussion of the agent transfer protocol. Since an agent and host can use any language, it is imperative that a protocol exist to facilitate agent migration. Specifically, we discuss some of the issues involved with agent migration and illustrate how the DADS solves some of these issues using functionality provided by the module system. Last, a development strategy for the transfer protocol is presented.

In addition to the development strategies discussed for each part of the DADS, Chapter 3 provides a detailed discussion of its current implementation. This includes an in-depth view of the dispatching system and state-machine implementations that make the overall DADS daemon, and module system work.

Chapter 4 presents a simple example to illustrate the operation of an agent in the context of the DADS system. In particular, it presents a detailed look at a simple intrusion detection agent executed within the DADS infrastructure.

To conclude, Chapter 5 presents a summary and some proposed advancements to the DADS.

CHAPTER 2

DADS DESIGN

There are many issues to consider in the design of a new agent system. For some systems, speed is preferred over portability. Similarly, security may be more important than speed. Regardless of how the underlying system is optimized, it must facilitate the execution and migration of agents across networks. For migration to occur between unrelated network domains (i.e., corporations, universities, etc.) the agent system must be portable enough to execute on a variety of architectures. Further, it must use a security model that can guarantee a secure environment for all entities involved (i.e., agents, hosts). In general, an agent system addresses the issues specific to the applications they are designed for. For instance, agent based e-commerce favors security. Likewise, scientific computing would benefit from fast and efficient agents. To address these issues, the DADS is designed to offer an agent system that facilitates agent execution and migration, while at the same time offering a customizable platform that can adjust to fit the needs of its applications. This chapter describes the organization of the DADS, focusing primarily on the major parts that contribute to its operation.

DADS Organization

The DADS is designed to accept a heterogeneous blend of agents on a variety of hosts that span unrelated network domains. Illustrated in Figure 2.1, the DADS is made up of several interrelated parts, namely a daemon and a set of loadable modules. Described in further detail later, each part contributes to the overall operation of the DADS, allowing it to perform its primary goal, the acceptance and execution of mobile agents.

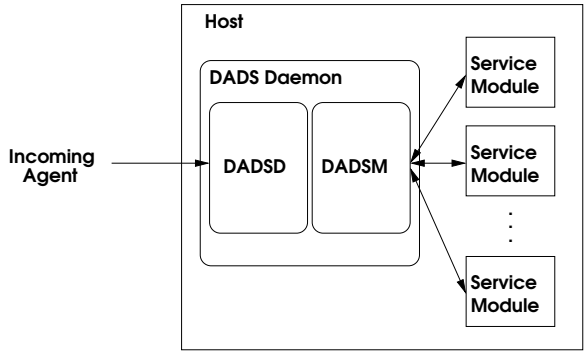


Figure 2.1: DADS Architecture

To participate in a mobile agent infrastructure, a host must provide an entry point for incoming agents. This requires a network access point, generally a TCP network port, which is understood by all agents and hosts that participate in the particular agent system. To achieve this, the DADS relies on a daemon process, labelled *DADS daemon* in Figure 2.1, which is designed to listen on a standard TCP network port for

incoming agents. As we discuss in Chapter 3, the DADS can be adapted to support other network protocols, not just TCP/IP.

In addition to network access, the DADS is also responsible for two other important functions. The first is the management of the service module system. Discussed in further detail later, modules provide a limitless library of services ranging from language availability and security to fault tolerance. Consequently, a host may offer any combination of services. That is, it is likely that different hosts on different networks will offer their own selection of services. Hence, an agent cannot presume certain properties (i.e., language, security) will be available on every host that it attempts to migrate to. Since an agent is autonomous, it must be able to determine whether the available resources on a remote host will allow it to execute after it migrates. Therefore, the second function of the DADS is to provide a transfer protocol (TP) to facilitate this decision making process. An integral part of the DADS system, the TP is an important mechanism in the acceptance and execution of heterogeneous agents.

Just like any other agent system, the DADS has its own definition of an agent. Since the DADS supports a wide range of services through its module system, it can potentially support a wide range of agent languages and functionality. Consequently, for an agent to take advantage of these available services, it must be flexible in its implementation. That is, the agent must be able to support a heterogeneous blend

of data, regardless of the structure and content. To do this, DADS agents are similar to the agents used in TACOMA, hence, they act as descriptive containers which alleviates the limitation on content. Described later, a DADS agent facilitates code mobility for a wide variety of languages.

The DADS is heavily influenced by the design of today's agent infrastructures. Instead of contradicting the many ideas that have been introduced by these systems, the DADS combines aspects of these concepts into a single agent platform.

DADS Agent Design

Traditionally, there are many different definitions of an agent. For AgentTcl, an agent is a state-captured image of an interpreter context, whereas Aglets and Mole agents are suspended JVM threads. TACOMA agents are high-level Tcl Procedures. Regardless of the system, the common property shared among the agents is that they contain a set of instructions and data. The instructions define the actions and behavior of an agent, whereas the data stores agent state.

Without a supporting infrastructure, agent mobility is impossible. In AgentTcl, agents cannot transfer unless they migrate to hosts running the AgentTcl system. Likewise, Telescript agents are unable to execute remotely unless a Telescript engine is available. In general, agents are dependent on their infrastructure for mobility and

execution. Frequently, mobility is achieved through special functionality that is built into the infrastructure core. Thus, when an agent requires transfer, it calls functions that package and transmit an agent to another host. In general, this functionality is placed directly into a language execution environment (i.e., modified virtual machine), thereby limiting the agent language to that of the execution platform. This creates an agent that is subject to all of the (dis)advantages of that particular language platform (i.e., speed, security, portability). Consequently, dependencies lead to niche infrastructures that work solely with agents of their own design.

In the context of DADS, an agent uses a design that is influenced by the TACOMA system. Expanding on its briefcase and folder concept, DADS agents act as containers (i.e., briefcase) for three distinct segments (i.e., folders). Unlike TACOMA where a briefcase can contain any number of folders, a DADS agent always contains the following three segments:

Code: Stores instructions that are used to determine the behavior of an agent. This segment can contain any arbitrary sequence of bits. Thus, it can store any language, byte-code, or state-captured image.

Data: Stores state information and gathered data used by an agent during execution. This segment may also contain any arbitrary sequence of bits.

Properties: Stores a description of the code and data segments. Thus, the properties (i.e., language, security, fault-tolerance requirements) of an agent can be identified.

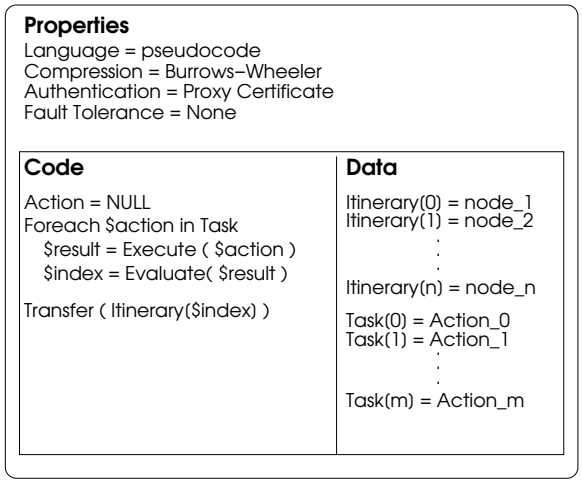


Figure 2.2: A DADS Agent

Further illustrated in Figure 2.2, code and data segments are complemented by the descriptive information found in the properties segment. This properties segment is very important to the operation of the DADS. Since the DADS does not limit the contents of an agent's code and data segments, DADS agents are free-form. This means agents can use any format for their language and data. When they are mobile, agents become free-form autonomous entities. If they are used across heterogeneous networks, it is impractical to assume that computing resources on every host are the

same. Nevertheless, an agent must be able to migrate, hence, it must be able to determine whether it can execute on a remote host. Without some form of identification, an agent cannot compare the available resources on a host with its requirements for migration and execution. To address this, DADS agents are equipped with a properties segment. Similar to a badge in Mole, the properties segment enables an agent to have knowledge about itself (i.e., language, security, fault-tolerance requirements). When used in conjunction with a transfer protocol, this knowledge allows us to exploit the specifications of an agent in terms of its requirements and its capabilities. As a result, an agent can map knowledge of itself onto its knowledge of a host (i.e., language platforms, security, fault-tolerance mechanisms). This allows it to make an informed decision about whether it can migrate and execute on a remote host. Similarly, when a host requires authentication, this properties segment allows the host to be able to understand whether the agent is capable of supporting the authentication it requires. Further, it avoids an agent format that is dictated by the implementation details of the infrastructure.

Properties

In contrast to the format used with the code and data segments, the format of the properties segment is fairly restrictive. A restrictive format is imperative since an

agent must be able to reference knowledge of itself quickly. Further, the format must facilitate a concise description that can succinctly describe all relevant information about the agent. To do this, the DADS uses a descriptive element which we refer to as a *property structure*. In general, a property structure contains four distinct elements:

Property Label: An identifier which labels the property as a capability (c), requirement (r), or both (b). This field facilitates the construction of requirement and capability sub-trees used with the transfer protocol.

Property ID: A standard number used to refer to a particular service. This number could be defined according to a local administrative domain or it could refer to a standardized number similar to the management information base (MIB) used for SNMP [6]. When moved to a global scale, this would require a globally accepted standard defined by a centralized unit, such as the Internet Assigned Numbers Authority (IANA) [14].

Property Name: A string used to describe the particular property. For instance, the Perl interpreter may use a Property Name of "Perl". In general, this would be used whenever the property ID cannot be matched.

Sub-Properties: A list of properties that further describe the particular property. For

instance, an agent may have an "Interpreter" property that consists of the sub-property "Perl". Further, the "Perl" sub-property may contain a "Version" sub-property.

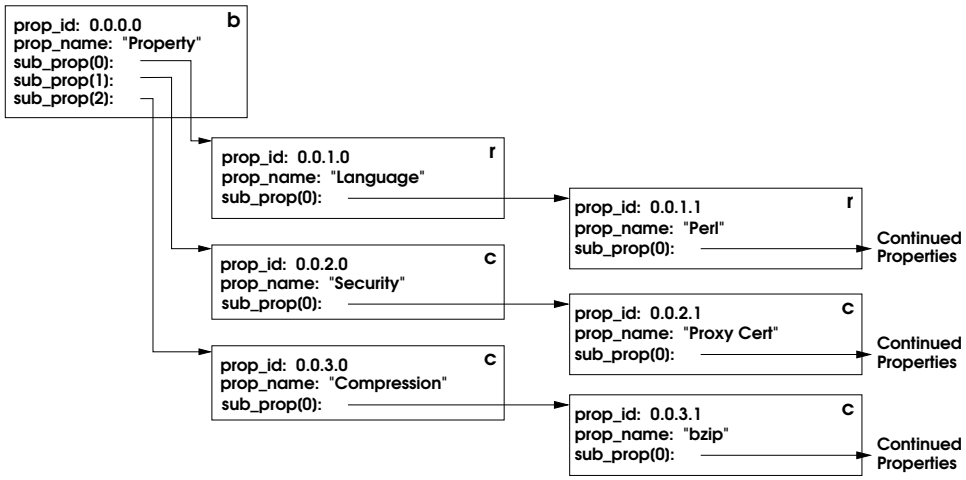


Figure 2.3: Property Hierarchy

Further illustrated in Figure 2.3, the property structure uses its sub-property element to form a hierarchical description of an agent, which we refer to as a *property hierarchy*. In this particular example, the agent's description is simple. The root represents the *Property* property from which a *language*, *security*, and *compression* sub-property are contained. As we descend the branches of this hierarchy, the agent description becomes more specific.

In addition to the description of an agent, property structures are also used by the DADS daemon to describe its loaded modules. Thus, property structures can be exchanged to determine whether agents can migrate and execute on remote hosts. Since an agent and a remote DADS-enabled host would use the same hierarchical format, it is possible to execute fast search algorithms to determine whether an agent's property hierarchy can match a subtree in the hosts property hierarchy.

In general, a property hierarchy contains descriptive paths that describe the *requirements* and *capabilities* of an entity (i.e., host, agent). In the context of the DADS, a requirement is defined as the functional service that is needed by an entity to successfully perform its tasks, whereas a capability is defined as a service that an entity can support. For instance, if an agent is written in the Perl language, it needs a Perl interpreter to execute, hence, the agent property hierarchy would contain a path in the property hierarchy that specifies the Perl interpreter as a requirement. Juxtaposed, a host does not need the Perl interpreter to execute, however, it needs to know that this service is available for agents to use. As a result, the host's property hierarchy would contain a path in its property hierarchy that indicates the Perl language interpreter as a capability.

Similarly, it is common for a host to use a form of authentication that will verify an agent's authenticity. In order to perform its task, an agent may not need to

authenticate, however, it is imperative that the agent understand that it can support a form of authentication if it is required by a host. Consequently, the agent's property hierarchy will contain a path to a capability that indicates that it can support the desired form of authentication. Likewise, a host may want authentication to occur before an agent is allowed to execute, hence, it will contain a path to a requirement that indicates the form of authentication that must be satisfied.

In the context of a property hierarchy, the delineation between a capability and a requirement is determined by the agent and host. That is, when an agent engages in the transfer protocol, it sends a subtree of properties derived from the paths in the property hierarchy that it has marked as capabilities. When this subtree is received by a host, it is compared to the requirements subtree on the host. This allows the host to select, from the agent's capability tree, the service it wants the agent to use. Similarly, after the host has made a decision, it sends its capability subtree to the agent, thereby allowing it to decide which service (i.e., authentication) it would like to use. In general, an agent and host do not send their complete property hierarchy across the wire, instead they send subtrees that have been derived from it.

Development Strategy

A DADS agent is best described as a mobile container for code and data. It provides a containment structure used to organize and describe arbitrary sequences of code and data. A DADS agent is separated into three segments. As discussed above, each of these segments hold a piece of information that is integral to an agent's operation. Since there are no limitations on code and data, a DADS agent provides a general vehicle for code and data transfer. This generality motivates a simple implementation. Illustrated in Figure 2.4, we present the basic format for an agent as it is used within the DADS infrastructure. As the code exhibits, an agent contains three distinct pointers, namely *Code*, *Data*, and a list of *Property* structures. In our initial implementation, the property structure has been kept relatively simple. If further attributes are needed to describe a property, they can be added. For now, we have included the four fields: a label *label*, a property id *prop_id*, a property name *prop_name*, and a property pointer **sub_prop* which facilitates the creation of a property hierarchy.

It is interesting to note how simple a DADS agent is. As discussed above, one of the main goals is to avoid an agent that is far too dependent on its underlying infrastructure. An agent system should facilitate mobility without placing a high number of restrictions on its agents. As Figure 2.4 shows, a DADS agent offers an

```
/* A Property Structure */
typedef struct service_prop {
        enum { B,C,R } label; // b = both, c = capability, r = requirement
        long prop_id;
        char prop_name[MAXNAMELEN];
        struct service_prop *sub_prop[MAXSUBPROPS];
} Property;

/* An Agent */
class Agent {                                                          10
        public:
                enum agent_field { CODE, DATA, PROPERTY };
                void *getField(agent_field af);
                void setField(agent_field af, void *val);

        private:
                int PropLength;
                int CodeLength;
                int DataLength;
                Property *Properties;                                  20
                char *Code;
                char *Data;
};
```

Figure 2.4: Agent Structure

interface to the contents of the agent without limiting its contents. This allows any segment of the agent to be easily referenced when it is needed.

The structure illustrated in Figure 2.4 is transparent to the user of a DADS agent. Instead of writing the agent to fit this format, a user writes an agent in the language of his/her choice and uses a special program called an *injector* to bootstrap an agent into the DADS system. Consequently, this program creates the structure described in Figure 2.4 by combining the code, data, and description of the agent. In addition, the injector transmits the agent according to the transfer protocol to a remote DADS enabled host for execution.

DADS Daemon

In order to participate in an agent system, a host must provide an entry point (i.e., TCP port) for agent acceptance. In addition, it must facilitate agent execution. In Telescript, a virtual machine provides this functionality, handling both agent migration and execution directly. In contrast, AgentTcl uses a dedicated process (*agentd*) which is responsible for accepting new agents and delegates them to independent processes (i.e., interpreters) for execution. In general, a host participates in an agent system by executing a daemon that can accept incoming agents. This daemon may be directly involved in the execution of the agent, or it may off-load the execution

somewhere else. Regardless of how the agent is executed, it is the responsibility of the daemon to define how the resources (i.e., languages, security, etc.) are used.

In a centrally managed network, node homogeneity is achievable. Since each host is administered by a central authority, installation of a uniformly configured agent system is possible. As discussed by Karnik and Tripathi [17], *"In a completely closed local area network...it is possible to trust all machines and the software installed on them"*. This level of trust is lost when an agent system is distributed across heterogeneous networks. Since different institutions may employ orthogonal policies (i.e., security, resource usage), it may not be possible to install a uniform configuration across all hosts. Therefore, it is imperative that an agent system allow the network administrator(s) to customize their hosts according to their own policies.

There are several issues to consider in the design of the DADS daemon. First, it must be able to accept and execute migrating agents from remote hosts. When we consider how general the DADS agent is, it is clear that an agent may contain any form of code and data. Hence, the DADS daemon must be able to handle any of the agent languages of its constituent agents. And second, it must provide a customizable platform that can adjust to the policies of an institution. To do this, the DADS daemon employs a design where the functionality for migration and language execution are separated. Parallel to AgentTcl, this design allows the DADS daemon

to focus on agent migration, allowing language execution to be off-loaded to other processes.

Development Strategy

The main focus of the DADS daemon is to provide two services, namely the acceptance of agents and the management of modules. Both of these services require a clean and efficient interface that can handle high volumes of agent interaction. Further, in order to run as a system level service, the final product must be simple.

The implementation of the DADS daemon is focused on an object oriented design. Built using the object model found in Figure 2.5, the DADS daemon is organized such that higher level objects provide simplistic interfaces to lower-level functionality. That is, higher level objects are *controllers* of their encapsulated objects.

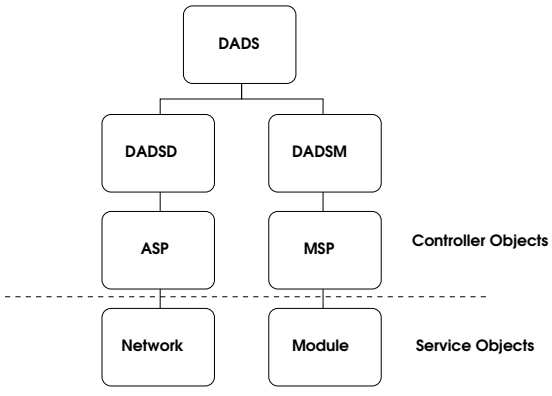


Figure 2.5: DADS Daemon Object Hierarchy

Located at the bottom of the object hierarchy, *service* objects encapsulate the low-level functionality specific to a particular management function. Labelled *Network* and *Module*, these objects provide an interface to functionality such as connection handling, name resolution, and I/O. Hence, it is within these objects that the code for socket manipulation (i.e., pipe, accept, read, write) is found.

At the next level up, we encounter two intermediate controller objects, namely the Agent Service Protocol (ASP) and Module Service Protocol (MSP) objects. It is within these objects that the protocols specific to DADS (i.e., agent delivery, module loading) are defined. In particular, methods provided by the Network object are used by the ASP object to execute the delivery protocol. Similarly, methods provided by the Module object are used by the MSP object to execute the loading protocol. As Chapter 3 explains, the ASP and MSP objects encapsulate the state-machines that facilitate execution of the particular protocols.

Next, the ASP and MSP objects are placed within a DADS Delivery (DADSD) and DADS Module (DADSM) object, respectively. As an added layer of control, the DADSD and DADSM objects provide a layer of abstraction where a list of ASP and MSP objects can be managed. This is advantageous for future development since we can derive other ASP objects with services other than the agent transfer protocol. For instance, we could create an ASP object that listens for management

data, or an ASP object that interacts with remote DADS daemons to handle faults. Further, this allows the DADSM to maintain functionality that could be used to do dynamic shutdown and loading of modules in the event that a service has been idle for an extended period of time. In general, this added layer of abstraction separates the dispatch functionality of the DADS object from the more in-depth protocol level functionality provided by the ASP and MSP objects.

Finally, the top of the object hierarchy contains the DADS object. Further discussed in Chapter 3, this object is designed to dispatch service among its member DADSD and DADSM objects. It is designed to wait for a request (i.e., connection, data available) from both the network and from the modules, and dispatch service accordingly. Consequently, the DADS object is responsible for calling the appropriate member functions within the DADSD and DADSM to handle requests. They, in turn, call member functions that perform the specific protocol defined within the ASP and MSP objects, which use the functionality provided by the Network and Module objects, respectively.

Modules

As discussed above, the DADS daemon is responsible for the management of its service modules. A module is a process that is started by the main DADS daemon,

hence, it can communicate with the DADS daemon using various forms of interprocess communication (i.e., pipes, FIFO, sockets). A module can read data sent to it by the DADS daemon from a standard input channel (see Figure 2.6) and it can use its standard output channel to write data back. Consequently, a DADS module is designed to read data, perform a function on that data (i.e., agents), and possibly rewrite the processed data back to the DADS daemon.

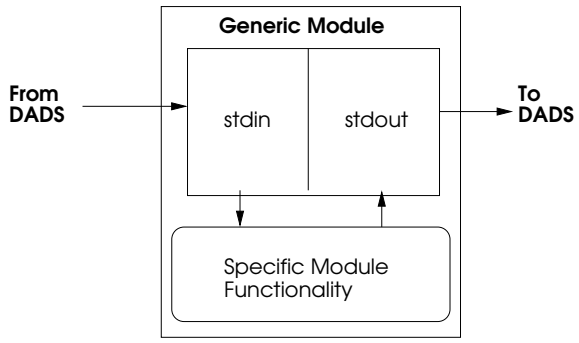


Figure 2.6: A Generic Module

When a module is started, it is not immediately recognized by the DADS daemon. Instead, the module must engage in a loading protocol, thereby providing a description of the service offered. In general, the loading protocol facilitates the construction of a property hierarchy that is recorded within the DADS daemon. Further, the loading protocol facilitates integrity checks to ensure that a module is authentic.

When a module is loaded, it is described using the loading protocol, thereby facilitating the construction of a property hierarchy. For a host, this property hierarchy contains a root property, referred to as *Property*. Similar to the agent's root property, this property serves as an initial search point for property comparison. As modules are loaded, they are registered as sub-properties of the host's root property. Thus, if a security module is loaded, it is loaded as a *security* sub-property and could contain further sub-properties that are more specific to that particular service. Likewise, if language and fault-tolerance services are loaded, they would be included as *language* and *fault-tolerance* sub-properties, respectively.

Module Chains

A major advantage of modules is realized through a DADS-specific concept referred to as *chaining*. Thus far, a module is described as a single process that interacts with the DADS daemon. With chaining, a module can interact with other modules, using one of two methods. The first method, illustrated in Figure 2.8, allows one module to load another module. For instance, a fault-tolerance module may load a language module. In this example, the fault-tolerance module receives data directly from the DADS daemon from its *stdin* where the data is processed and immediately written to *stdout*. The language module, which reads its *stdin*, receives this data for

execution. Consequently, the module chain forces the data to be pre-processed by the fault-tolerance module before it is handed over to the language module for execution.

In contrast, the second chaining method allows each module to be loaded independently by the DADS and the loading protocol is used to indicate that a chain exists. In the example above, the loading protocol would flag the language module as being dependent on the fault-tolerance module. This dictates that the data intended for the language module must first be sent through the fault-tolerance module. Since the two modules input and output channels are not connected, they are unable to talk to each other directly. Instead, the modules route the data back through the DADS daemon. Hence, incoming data is sent to the fault-tolerance module first. Once it is processed, the data is sent back to the DADS daemon, which then routes the data to the appropriate language module for execution. In general, chaining is limited only by the imagination of a module's developers. Further, module implementations should be kept simple, whereas chains should be used to create more complex functionality.

Module Functionality

As independent processes, modules allow the DADS to offer concurrent services. This is advantageous since there are many issues to address in an agent system [17]. For the DADS, we can build a module to address each of these issues, however, in

this thesis we focus primarily on the following:

- Security
- Fault Tolerance
- Language Availability

Security Mobile agents require a secure environment that can guarantee that an agent is safe from data tampering during both execution and migration. In contrast, hosts must be able to trust that incoming agents will not pose a serious threat to the host (i.e., cause a crash, destroy files, etc.). In general, both the agent and host must be safe from malicious action, hence, an important aspect of any mobile agent system is security. Fortunately, a great deal of research is being done in this area [9, 15], thereby offering several philosophies regarding how agent security should be managed. Even though the design and implementation of these philosophies far exceed the scope of this thesis, it is important to recognize that an agent system must be able to incorporate new ideas as efficiently as possible. To accomplish this, the DADS module system is designed for quick integration of the latest technologies, including security. As illustrated in Figure 2.7, the DADS can handle any type of security service, given that the module is written according to the design principles stated above.

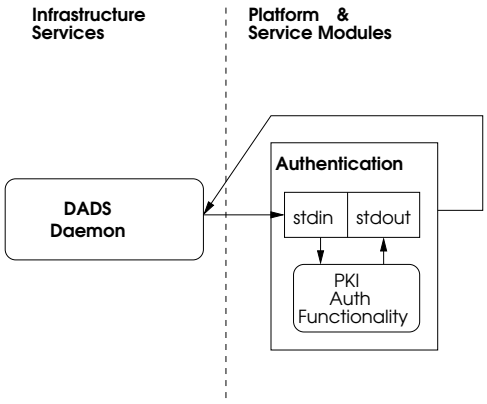


Figure 2.7: Security Modules

It is important to authenticate agents before they are allowed to execute. If an agent does not authenticate, a host cannot determine an agent's source. Further, if the agent is malicious, it is imperative that the agent be traceable back to its author. Simple issues such as these are what drive an agent system to support a method for agent authentication. For the DADS, authentication can be realized using one or more authentication modules. This may include methods that support public-key cryptography, etc.

In addition to authentication, the integrity of an agent is extremely important to the success of the agent paradigm. Unless there are mechanisms designed to prevent it, agent tampering (i.e., modification of an agent) is an effective way to destroy the integrity of an agent. In particular, no entity (i.e., host, another agent) should be able to modify an agent's contents without its permission. In general, it is difficult

to prevent tampering, however, it is relatively simple to detect when a change has occurred [23]. Therefore, it would also be advantageous for DADS to include a set of modules that can detect when agent tampering has occurred.

Any agent system must provide a robust model for security. In AgentTcl, certain aspects of agent security are addressed using PGP [13]. Other systems, such as TACOMA, employ a security model based on the concept of electronic cash [16]. In general, agent systems use security models that best fit the applications they are designed for. The DADS is no different, yet its modular design allows additional security mechanisms to be loaded as they are needed.

Fault Tolerance In addition to security, agents must be able to recover from fatal errors. Since an agent executes code on a remote host, agent failure may introduce data loss. Worse, fatal error could destroy an agent, losing it altogether. Nevertheless, if agents cannot survive in a relatively hostile environment (i.e., the Internet), the success of agent based computing is hindered. Similar to its dependence on a platform for execution, an agent is fully dependent on its underlying infrastructure for fault tolerance. However, just as there are modules to support several mechanism for security, a module like the one shown in Figure 2.8 can be added, thereby adding mechanisms to address fault-tolerance.

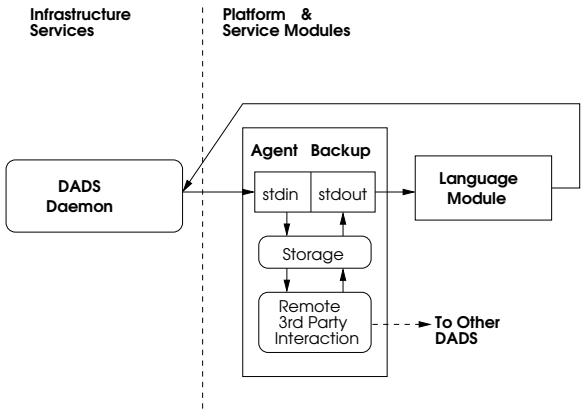


Figure 2.8: Fault Tolerance Module

To further illustrate the need for fault-tolerance, consider an agent that has executed for several hours gathering stock information in order to make buy/sell decisions. If this agent migrates to a host experiencing hardware problems, there is a high probability that a fatal error will occur. If an error occurs, the agent along with all of its gathered data could be lost. From a user standpoint, fatal errors could directly result in monetary loss.

There are several solutions that could be integrated into a module. One solution suggests that it may be possible to require the agent to send a backup of its data, at regular intervals, to a stable and secured host. However, a problem exists if the agent is delayed and not terminated. If timeout mechanisms are used, the host on which the agent was created may consider the delayed agent as dead. In response, the host

might create a new agent to pick up where its predecessor left off. Generally, this new agent will probably follow a strategy similar to the original agent. Thus, when the original agent is no longer delayed, two agents will be working on the same problem, which could result in an inconsistency.

In general, it is the responsibility of the infrastructure to recover from agent error. In the example above, neither the agent nor the infrastructure is capable of detecting a hardware failure. However, if the infrastructure communicates with other agent systems, crashes and agent terminations could be monitored.

There are many techniques that could address the issue of fault tolerance. Instead of implementing all of them, the DADS allows certain fault tolerant modules to be loaded. When used in conjunction with the transfer protocol, agents can better decide whether the available level of fault-tolerance can guarantee the survival of the agent.

Language Availability An agent infrastructure must provide a language environment to execute its constituent agents. Since many platforms exist, it is impractical to assume that a multi-lingual agent system will support all known languages. Nevertheless, to successfully execute agents, the agent system must make a subset of these languages available. As discussed above, data is transferred from the DADS daemon to its modules through standard input. Illustrated in Figure 2.9, several

language modules can be available concurrently, making the DADS a multi-lingual agent system.

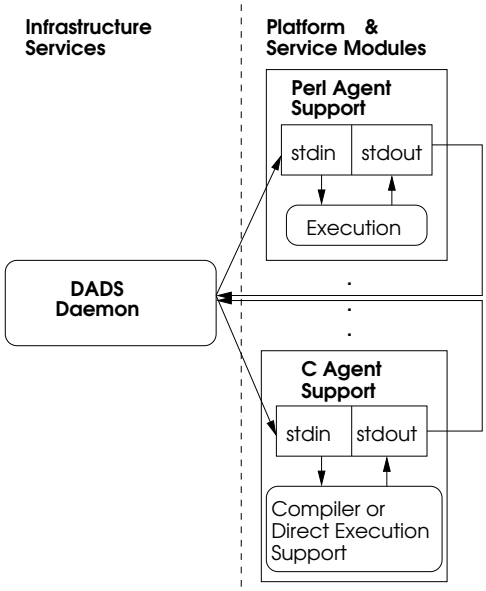


Figure 2.9: Language Modules

In the DADS, language modules act as wrappers for more advanced language services. In general, a module that is geared towards language availability will commonly provide one or more of the following services:

Compilation: An agent is sent to the module from the DADS daemon as textual C code. This code may require compilation before it is executed. Generally, code compilation is used where execution speed is a concern and mobility is infrequent.

Interpreted Execution: An agent is sent to the module from the DADS daemon as textual code or as a snapshot of an interpreter context. Most similar to the agents used in TACOMA and AgentTcl, these agents are generally executed by an interpreter.

Direct Execution: An agent is sent to the module from the DADS daemon as a snapshot of an operating-system process. These agents are highly dependent on the underlying host architecture and are commonly used in environments that are under a single administrative domain (i.e., LAN).

It is important to note that a language module is not limited to one of the above services. If developers require other services, they are free to add a new module that fits the needs of their application. However, as soon as a module is specialized for speed (i.e., direct execution), the number of hosts that can execute agents within this environment are limited. Likewise, when a module is designed for portability (i.e., interpreted execution) the speed of execution is degraded. Consequently, a tradeoff exists between speed and portability. In other agent systems, this tradeoff is typically decided by the system developers at build time.

In contrast, the DADS is a customizable agent system that allows the system administrator(s) to decide at run-time, how resources are used to facilitate the agent

paradigm. Using modules, the DADS is easily modified to support speed and/or portability when it is needed. Further, since there are no limitations to the services that can be offered as modules, the DADS acts as a portal to resources.

Development Strategy

The requirements of a module are simple. To be used with the DADS daemon, a module must provide read and write capability on a standard input and output channel, respectively. In addition, the module must be able to engage in a basic protocol that allows the DADS daemon to load, unload, and manage the module. As long as a module supports these functions, it can be written for any purpose. Consequently, it is impossible for this thesis to list every possible path to building a module, thus, we present a basic strategy for a module's construction.

As discussed above, the code segment of an agent can take many forms. In particular, agent instructions can be text or they can be contained within a state-captured image. Regardless of the format, the ultimate goal of an agent's code is execution. Therefore, a module must be loaded to execute the code of the agent. Illustrated in Figure 2.10, we present a simple module which reads and executes agents written in Perl.

To begin, the module code illustrated in Figure 2.10 loads an agent functionality

```
#!/usr/bin/perl

use AF;

AF::Register();

while(1){
        agent = AF::Read();
        AF::Process(agent);
}
```

Figure 2.10: A Basic Perl Module

(AF) library which contains the functionality required to interact with the DADS. Described later, libraries like this facilitate a concept referred to as *platform migration*. In addition, the library may contain functionality, such as status routines, that may be useful to an agent. Once the library is loaded, the module registers itself with the DADS using the *AF::Register* subroutine. This function engages in the loading protocol, which sends a description of the module (i.e., property structure) to the DADS. Once it is registered, the module enters an infinite loop where it calls and blocks on the *AF::Read* subroutine. When an incoming agent is sent to the module, it records a reference to the agent in the *agent* variable. Next, this reference is used as an argument to the *AF::Process* subroutine, which ultimately executes the agent code. For this type of agent, the code could be executed using the Perl *eval* function. If required, modules could also support mechanisms for state-capture. In

this environment, the module may use a design parallel to AgentTcl where a modified interpreter reads state-captured images from standard input.

In the example above, we ignore several important issues. First, there are no mechanisms built into the module for multi-agent execution. When an agent is sent to the module, it is given full control of the interpreter. This means that any new agents will have to wait until the module has finished executing a prior agent. Further, this module offers absolutely no security. Since it is using the normal Perl interpreter, the agent can perform any function that is available in the Perl language. In general, modules will use more advanced techniques than what we have displayed, however, the basic principles stay the same.

Infrastructure Protocol

A free-form agent presents an interesting dilemma when it migrates. The DADS cannot assume that certain resources are used by an agent. Likewise, an agent cannot assume that the DADS has made certain resources available on a host. Therefore, before migration can occur, it is imperative to have a facility that can communicate the properties of both agent and host. To accomplish this, the DADS uses a special transfer protocol (TP).

The TP is a multi-step exchange that is executed between an agent and a remote

DADS daemon. As discussed above, agents require the availability of certain language platforms, forms of security, and/or fault-tolerant mechanisms. Further, agents may use a data format (i.e., compression) that requires pre-processing before the agent can be executed. From a host standpoint, security is extremely important. In a secure setting, a host may require an agent to verify, authenticate, and/or pass integrity checkpoints before the agent is allowed to migrate. In general, the requirements of both an agent and host must be met before migration can occur. Since a DADS agent contains a properties segment, it is aware of what it must communicate. Similarly, the DADS manages the loaded modules, hence, it is aware of the services it provides. When an agent is ready to migrate, it contacts a remote DADS enabled host and engages in a dialogue according to the TP. Figure 2.11 illustrates this process.

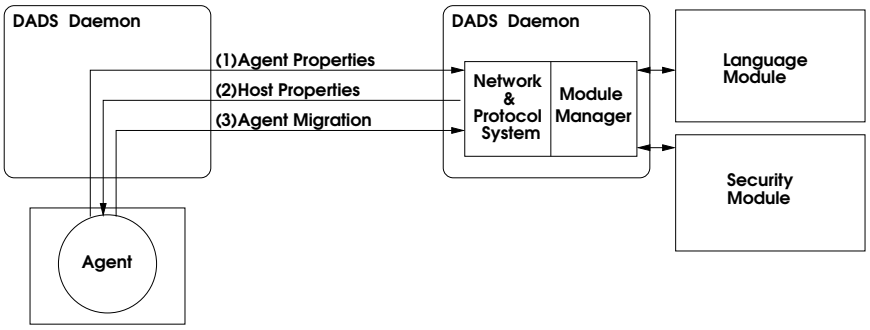


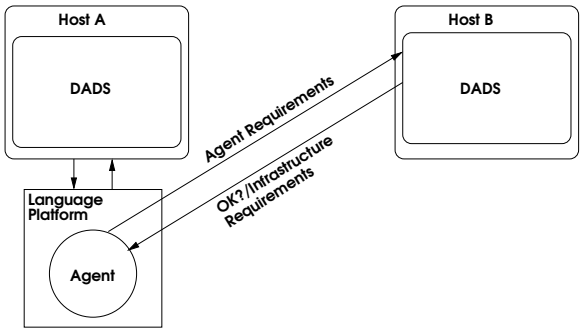
Figure 2.11: The Transfer Protocol

In general, there are three phases to the TP. The first phase, labelled (1) in Figure 2.11 occurs when a connection is made to a DADS daemon. Once connected, an agent

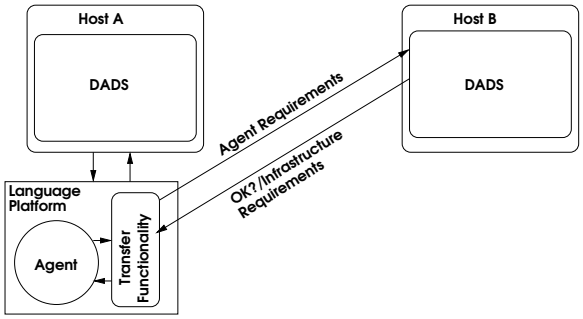
transmits a subset of the information stored in its properties segment. This is received by the DADS, allowing it to decide which services, if they are available, it would like the agent to use. If the DADS is unable to support the agent, the TP protocol dictates that the DADS return a negative reply and terminate the connection. On the other hand, if the DADS can support the agent, the TP moves to phase (2). At this point, the DADS replies with a property hierarchy that describes its selections from phase (1). In addition, the host transmits its capabilities (i.e., authentication methods, etc.). Similar to host's decision making process, this information allows the agent to decide which services, if they are available, it would like the host to use. Finally, if the agent supports the host's requirements, the TP moves into the agent migration phase (3) where authentication, and the final transmission of the agent takes place.

Similar to how an AgentTcl agent executes, a DADS agent executes within the context of a language module. If an agent is written in Perl, it will execute within a module that supports the Perl language. Likewise, if the agent is written in C and requires compilation, a module is written to provide this service. If state-capture services are needed, modules can be written with state-capture support included. Regardless of how an agent executes, it can engage in the DADS TP using one of three methods.

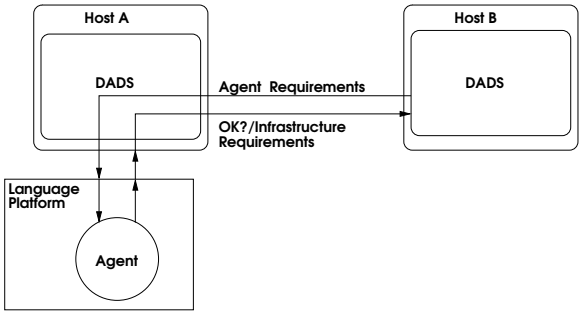
The first method, shown in Figure 2.12a is referred to as *direct migration*. This



(a) Direct Migration



(b) Platform Migration



(c) Proxy Migration

Figure 2.12: Migration Methods

method requires an agent to carry the code used to perform the TP. Thus, instead of using special functionality provided by its language platform, agents reference their own functionality. This avoids having to modify a language platform. Further, it increases the self-sufficiency of the agent. However, direct migration increases an agent's size, making this form of migration less efficient.

The second method, shown in Figure 2.12b is referred to as *platform migration*. This method is most similar to the forms of migration found in today's agent systems. Using a modified language platform, agents make function calls to special subroutines that perform the TP. By moving the migration functionality into the language environment, agent size is reduced, hence, the agents are more efficient. In general, platform migration is the fastest migration technique, however, it may require the modification of a language platform.

Finally, the third method, shown in Figure 2.12c is referred to as *proxy migration*. This method addresses the burden of extra code in direct migration. In addition, it addresses the modified language environment requirement in platform-migration. In this method, the TP is integrated into the DADS daemon. As discussed above, an agent executes within the context of a language module. Further, all DADS modules have the ability to interact with the DADS daemon. This allows a module to send and receive data through an interface connected to the DADS. Proxy migration takes

advantage of this interface, allowing an agent to send its image and data back to the DADS. Thus, when an agent decides to migrate, it can contact the daemon and send an agent image along with a destination address. The daemon then contacts a remote DADS and performs the TP. If the transfer is successful, the agent on the local host is terminated.

A tendency of today's agent systems is to use some variant of platform migration. While this facilitates faster and more efficient agents, it also tends to limit an agent to a single methodology for code mobility. With the DADS, agents can supply their own mobility functionality (direct migration), or use the DADS daemon to facilitate migration (proxy migration).

Development Strategy

Autonomy introduces a significant burden on an agent. As discussed above, a DADS agent is free-form and it is not limited to a standard system-wide language, security model, or fault tolerance system. Similarly, the DADS daemon is also free-form in that different hosts may provide different types of services. Therefore, we have developed the TP to facilitate agent migration to remote DADS hosts.

The TP is responsible for the exchange of property hierarchies between an agent and host. That is, it defines how to transmit a description regarding a host's available

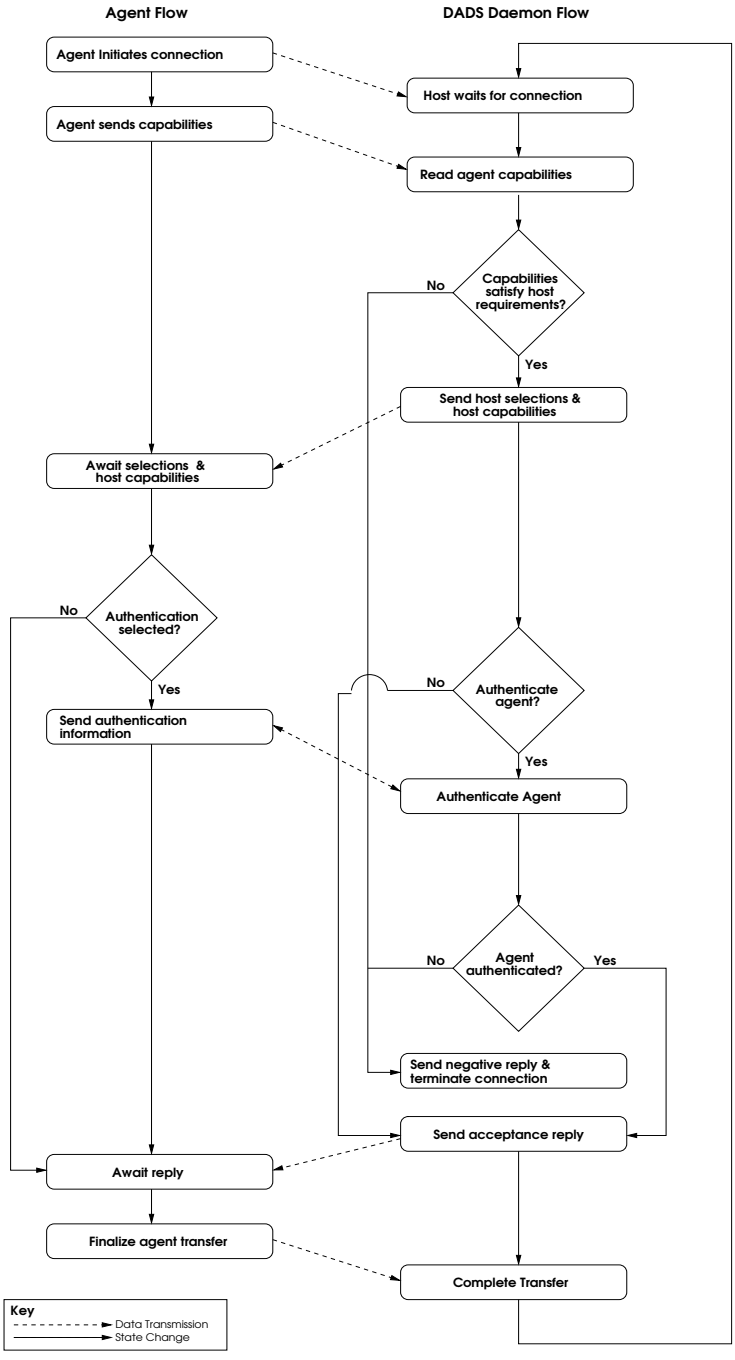


Figure 2.13: Transfer Protocol Flowchart

resources to an an agent and vice-versa. Since both an agent and host are aware of their properties, each can decide if migration should take place, based on the properties of the other. Described in detail by Figure 2.13, the TP does not define the content of the exchange. Instead, it defines the decision making process, which is used by an agent and host to make a migration decision.

Illustrated in Figure 2.13, the TP also includes mechanisms for authentication. In the DADS, an agent can migrate without performing any form of authentication. While it may pose a security risk, we feel that this, along with many other configuration decisions, should be left to a hosts administrator(s). However, if security is important, then authentication is facilitated by the TP. It is important to note that the TP does not define the authentication method. This is provided by a security module designed for that purpose. Instead, the TP calls the authentication service, which ultimately returns a positive or negative response. This reply alters the TP flow accordingly (i.e., accept or deny the agent).

In the context of the DADS daemon implementation, the TP code is encapsulated within the DADSD object. As discussed above, the DADSD is responsible for agent acceptance, hence, it follows that it is responsible for the TP. To be more specific, the TP code is further encapsulated within an ASP object (see Figure 3.1). In the initial DADS implementation, the ASP object is designed to handle agents using

the TCP/IP protocol suite. Further, it provides a simple interface that the DADSD object understands. The primary advantage of this is the expandability for future protocols. If support for future protocols is required, the protocol definition and functionality is simply placed in additional ASP objects. Therefore, as long as the ASP object provides a standard interface, protocols can be added without having to modify our core daemon code.

CHAPTER 3

IMPLEMENTATION DETAILS

The focus of the DADS implementation is simple; an efficient, low resource footprint, system level application that can handle high numbers of incoming agents. Further, it must be portable and robust enough to facilitate future development. To accomplish this, we have used the C++ language to develop the major portions of the DADS system. Supporting object orientation, C++ is a language that can support the proposed development strategies discussed above. In addition, it avoids the overhead of using a virtual machine such as Java. This chapter discusses the details concerning the implementation of the DADS system. In particular, it discusses the DADS daemon and its dispatching system. Further, this chapter discusses the state-machines that control and perform the transfer and module loading protocols. And finally, it discusses the necessary functionality that is needed for a module to operate.

DADS Daemon

Probably the most important piece of the DADS system, the DADS Daemon is responsible for incoming agents and the modules that execute them. Illustrated

in Figure 2.5, the main DADS object is a container object designed to encapsulate both the network (DADSD) and module (DADSM) systems. Since these services are maintained in separate objects, it is also the responsibility of the DADS object to coordinate and control them. In other words, the DADS object is responsible for dispatching control to the DADSD and DADSM when they require attention (e.g., connection, data available). Hence, it is imperative that the DADS be able to detect when one of its constituent objects require attention. This is accomplished using a mechanism referred to as a *hook*, which, in its initial implementation is simply a file descriptor (i.e., socket). Illustrated in Figure 3.1, when a DADS object is instantiated, its constructor instantiates its member objects, *dadsd* and *dadsm*. Using the methods provided by these objects, the DADS object retrieves a set of hooks from its members, using the *ReturnHooks* method, which allows it to place the returned values into its member array labelled *hooks*.

The DADS daemon is designed to run as a single process, hence, it must be able to process data quickly and without any delay. Thus, if the DADS object is a single daemon that dispatches service to the network and module systems, then how does it give one side attention without introducing delays on the other? More specifically, what happens if a connection becomes idle while sending large amounts of data, while at the same moment a module requests service? If the logic waits until

```
class DADS {
        public:
                void ServiceLoop();

        private:
                HOOK hooks[MAXHOOKS];
                DADSD *dadsd;
                DADSM *dadsm;
};

class DADSD {
        public:
                void ProcessRequest();
                HOOK *ReturnHooks();

        private:
                ASP *asp;
};

class DADSM {
        public:
                void ProcessRequest();
                HOOK *ReturnHooks();

        private:
                MSP *msp;
};
```

Figure 3.1: DADS Objects

the connection has sent its full payload, the module may have to wait a significant amount of time (maybe forever) before it is processed. Consequently, this is where hooks are important.

Since they are descriptors, hooks can indicate when they are ready to perform I/O. If they are set non-blocking [25], the functions that use the descriptors will return if there is nothing to do (i.e., idle connection, full read/write buffer). When used in conjunction with a multiplexing system call such as *poll* or *select*, the DADS dispatch logic can sit and listen to the various hooks until I/O service is requested. Since the descriptors are non-blocking, system calls such as *read* or *write* will only return as much data as is available at that time. However, this comes at a price. Since the system calls can return without reading or writing a complete payload, it is the responsibility of the higher-level code to keep track of where the last operation left off. To address this, the DADSD and DADSM objects maintain ASP and MSP objects. These objects encapsulate independent state-machines that track the progress of the protocol for each module and connection.

In general, the DADS daemon begins to service requests (i.e., listen on network, load modules) when its *ServiceLoop* method is called. This function enters an infinite service loop which listens for I/O activity, via the hooks, using the *select* system call. If a request from the DADSD is detected (i.e., connection, data), the DADS processes

the request with a call to the DADSD *ProcessRequest* method. Likewise, if a request comes from the DADSM (i.e., load a module, module error, etc.), the DADS handles the event in a similar fashion. While it can be argued that the DADSD and DADSM objects are extraneous, their presence allows future developers to add ASP and MSP objects that encapsulate state-machines for services that supplement the transfer and module-loading protocols.

ASP

The ASP object provides the functionality used to engage in the agent transfer protocol. As such, it is responsible for setting up a server which listens on the network for incoming connections. In addition, since the daemon uses non-blocking descriptors, the ASP object must maintain state for each connection. To accomplish this, the ASP object maintains two very important member variables, namely a *TCPServer* object and a list of *Connection* objects.

Derived from the *Network* service object, a TCPServer object encapsulates all the functionality necessary to setup and maintain a TCP server. In addition, it provides an *Accept* method, which accepts a connection and returns a TCPClient object. Designed to make the DADS networking code clear and concise, the objects provided by the Network service object help to reduce code redundancy (see Appendix).

```
class Connection {
    public:
        TCPClient *incoming;
        Session *session;
        Agent *agent;
        int hook;
        enum { IDLE,
               CODE_SPEC,
               DATA_SPEC,
               PROP_SPEC,
               CODE_CONTENT,
               DATA_CONTENT,
               PROP_CONTENT,
               PROP_CHECK,
               AUTH_INIT,
               AUTH_ENGAGE,
               ACCEPTANCE
        } p_state;
        unsigned int b_read;
        unsigned int b_written;
};
```

Figure 3.2: Connection Object

In order to track the state of each connected client, the ASP object also maintains a list of connection objects. Illustrated in Figure 3.2, a *connection* object is used to record client-specific information, where each member variable is used for the following:

TCPClient *incoming: Tracks the network connection information, this is used to communicate with the client.

Session *session: When a client engages in I/O with a module, a *Session* is created. Thus, when a client sends data to a module, it uses the Session to determine the module it is supposed to write to.

Agent *agent: Information that is gathered from the client is placed within this Agent object (see Figure 2.4).

enum {...} p_state: Records the current state of the connection as it relates to the protocol of the particular ASP object.

unsigned int b_read: Tracks how many bytes have been read in the current state.

unsigned int b_written: Tracks how many bytes have been written in the current state.

As clients connect, new connection objects are created to track their progress through the transfer protocol, hence, these objects are maintained for the duration of the connection. If the client terminates the connection or is idle for an extended period of time, the corresponding object is removed from the list, thereby closing the connection.

With the connection data structure, it is possible to keep state for multiple connections and track their progress as they engage in the transfer protocol. In particular, state is needed since non-blocking code must assume that a connection is sending one-byte at a time (i.e., a latent connection). To better understand how the protocol works, Table 3.1 defines each state and discusses how they interpret the incoming data. It is important to note that at any point, the state can return to IDLE, which indicates that some part of the protocol was violated (e.g DATA_SPEC indicates an agent size that is too large), thereby causing the connection to be terminated.

A particularly interesting feature of the protocol is located in the AUTH_INIT and AUTH_ENGAGE phases of the protocol. Up to this point, an incoming agent has sent all three of its segments to the DADS, however, the agent has not yet been accepted. The reasoning behind this lies in the fact that some authentication protocols, particularly hashing and public-key mechanisms, may require that the agent information already be present on the host. That is, in order for the authentication

State	Next State	Description
IDLE	CODE_SPEC	New connections begin in an IDLE state. State transitions after the object has initialized.
CODE_SPEC	DATA_SPEC	State transitions when the size (4 bytes) of the code segment has been received.
DATA_SPEC	PROP_SPEC	State transitions when the size (4 bytes) of the data segment has been received.
PROP_SPEC	CODE_CONTENT	State transitions when the size (4 bytes) of the property segment has been received.
CODE_CONTENT	DATA_CONTENT	State transitions when CODE_SPEC number of bytes have been received.
DATA_CONTENT	PROP_CONTENT	State transitions when DATA_SPEC number of bytes have been received.
PROP_CONTENT	PROP_CHECK	State transitions when PROP_SPEC number of bytes have been received.
PROP_CHECK	AUTH_INIT or ACCEPTANCE	State transitions when the services are selected from the incoming property hierarchy.
AUTH_INIT	AUTH_ENGAGE	If authentication is required, state transitions when a positive reply is obtained from authentication module.
AUTH_ENGAGE	ACCEPTANCE	State transitions when agent has been notified that it has been accepted.
ACCEPTANCE	IDLE	Return to IDLE state, and terminate connection.

Table 3.1: ASP State Descriptions

to occur, the module needs to have access to the complete agent. If we were to delay the transmission of the agent until after authentication, then it is possible for a remote agent to provide false information and perhaps send contents that do not match the authentication. By sending the contents of the agent first, the particular module can use the local copy instead of having to trust that the remote agent will send valid data. Further, if interaction is required, the module can communicate with the remote agent and use the local copy for verification.

MSP

The efficient implementation of a module management system creates a very interesting problem. Since the DADS daemon can support simultaneous client connections, a problem arises when multiple clients want to use the services of a single module. For instance, several clients may need to participate in authentication. If only a single authentication module is loaded, only one client can use that service at a time. This is a problem since it not only degrades the performance of the DADS system, but it also potentially makes the DADS vulnerable to a denial-of-service attack. Therefore, it is imperative that the DADS employ a mechanism that can provide module service for more than one incoming client. To address this, we have created the MSP object.

In practice, when the DADS loads a module, it create an MSP object. This object

takes two parameters, namely a module name (i.e., filename to load) and a number which dictates how many copies of the module to start. By starting multiple copies of the module, the MSP object is able to manage a set of identical modules which can be used to service concurrent connections. Thus, when a connection requires service from a module, it creates a relationship between itself and one of the available modules managed by the MSP object. This relationship, referred to as a *session*, binds the client to the module for the duration of the client connection. By default, an MSP object supports only one session per module. That is, no other connections can interact with the module until the current session is terminated. In general, the single session per module approach is not a panacea to the concurrent connection problem (i.e., more connections than available modules); however, it provides a better solution than having only a single available module.

To fully address the problem, an MSP object also supports a multiple sessions per module mode. In this mode, only a single module copy is loaded, however, the module is written to handle concurrent connections. Therefore, when multiple connections require service from the module, separate sessions are created for each connection. These sessions employ unique identifiers that are used to tag data. Thus, when a module receives data from the DADS, it can use the tag to determine the source. While this may complicate the implementation of a module, it avoids the overhead

of loading multiple modules.

```
class Session {
        public:
                Connection *client;
                Module *module;
                enum { IDLE,
                       BUSY
                } s_state;
                MSP *msp;
};
```

Figure 3.3: Session Object

In order to track its sessions, the MSP object maintains a list of *Session* objects. Illustrated in Figure 3.3, a session object uses its member variables to achieve the following

Connection *client: Tracks the incoming connection so that information can be written back to the connected client. This is similar to how a connection object uses a session object to determine which module it needs to write to.

Module *module: Tracks which module the connected client is currently using.

enum {...} p_state: Records the current state of the session. Currently only two states exist: IDLE, which indicates that the module is available for use, and BUSY, which indicates that it is being used.

MSP *msp: Since a connection object maintains a session, this pointer allows the connection object to directly reference the MSP that this session is a part of.

As discussed above, the MSP object, by default, maintains only a single session per module. However, if the module is designed to support it, it can specify during the loading protocol, that the MSP object should use multiple sessions per module. Therefore, using these MSP objects, the DADS can provide a flexible interface that allows its modules to decide on how they are used.

Module

A module is an independent process that is attached, via a pipe, to the main DADS daemon. As such, the daemon must be able to organize module information (i.e., state, pid, pipe information), into a single object that can be easily managed by an MSP object. Introduced in Figure 2.5, the DADS daemon uses a *Module* service object to encapsulate this module information. Further, this object is responsible for loading and executing (i.e., fork and exec) an executable file. Therefore, when a module object is instantiated, it takes a filename as a parameter and creates a child process of the DADS daemon.

As soon as the child process is loaded, the code that has been started must immediately engage in the loading protocol. In general, the loading protocol is a two

step process, where the state of the protocol is maintained within the module object discussed above. The first step, registers the module with the DADS daemon. In particular, this step facilitates the merging of the module's property description into the daemon's main property hierarchy. Thus, in its most basic form, the loading protocol only requests that a module send its property description to the DADS daemon. For the loading protocol, this requires that a module object only keep track of two states, namely a LOAD and READY state. When this object is in the LOAD state, data coming from the child process is used for creating a property description. When the full property description is received by the daemon, the module object transitions its state to READY. At this point, incoming data from the module is either forwarded to a client engaged in a session, or it is forwarded to another module (i.e., module chain).

In practice, modules will most likely be written to load standard libraries that incoming agents can use. In the event that an agent uses a form of direct migration, no library is necessary since the agent contains all the functionality that it needs. However, as soon as an agent uses platform migration, the agent assumes that certain functionality is provided by the underlying execution platform. For a simple module like the one described in Figure 2.10, the code loads a library (AF) to make a set of functions and variables available for an agent to use when it executes. To better

understand what is provided by such a library, Table 3.2 gives a description of the functions and variables that are available to any agent that executes within this environment. For this particular module, we assume that the module supports the

Name	**Context**	**Purpose**
Code	Variable	Holds the agent code. Could be modified by the agent to create self-modifying code.
Data	Variable	Holds the agent data. Stores information such as itinerary and agent state.
Prop	Variable	Holds the agent properties string. Stores information such as itinerary and agent state.
Move	Function	Functionality used migrate an agent.
Register	Function	Registers the module with the DADS daemon.
Read	Function	Used to read an agent from the DADS daemon.
Post	Function	Used to write data, if necessary, to the DADS daemon.
Process	Function	Used to execute the agent (i.e., *eval*).

Table 3.2: AF Library Functionality

single session per module mode, as described above. That is, only one agent will execute in this environment at a time. Consequently, this allows the library to provide the global variables: Code, Data, and Prop which can be used by the agent to access its respective segments. When multiple agents are allowed to execute within a single module environment, certain aspects of the library must change.

CHAPTER 4

APPLICATION

The agent paradigm introduces an alternative method for solving problems in distributed computing. Agents add a layer of intelligence to the more traditional forms of remote programming, such as RPC and Mobile Objects. Further, agents provide a level of autonomy not realized in other forms of remote programming systems. When combined, these qualities can be exploited to offer effective solutions to many problems. In this chapter we provide an example application of the DADS. In particular, the following example is designed to illustrate the migration and execution of an agent, the operation of the transfer protocol, and the use of the DADS module system. It is important to note that this example illustrates the operation of a simple agent within a basic DADS environment, hence, we make certain assumptions about the network in which the agent(s) exist.

Application - Intrusion Detector

The security of a network and its constituent hosts is an important responsibility for any system administrator. Further, as the number of hosts on a network increase,

so too does the overhead of monitoring each system for a potential compromise. When network growth exceeds the monitoring abilities of its network administrators, some systems may be ignored in favor of more mission-critical ones.

In many ways, this problem lends itself to a solution founded in traditional client-server methods, where a centralized server listens as systems report, via the network, their current security status. If the status of these systems is polled frequently, a significant amount of network bandwidth may be wasted. To address this, the administrator(s) could employ an agent based solution. That is, instead of introducing large amounts of network traffic where clients repeatedly report normal status, an agent may be deployed which can migrate to each host and take appropriate action when abnormal status is detected. Illustrated in Figure 4.1, we present a Perl agent for the detection of an abnormal process. Also, for this example, it is assumed that each participating host has loaded the Perl module that has been discussed throughout this thesis. For this example, an agent of this type may also contain additional mechanisms for log-file analysis and file-system inspection. However, agent based intrusion detection is beyond the scope of this thesis, hence, we focus on a simple agent implementation to exemplify basic DADS principles.

In addition to code, the agent contains a segment for data. During execution, this data may change according to how the agent perceives its environment. Before

```
$hostname = `hostname`; // Retrieve the systems name
$process_list = `ps -axu`; // Retrieve a list of running processes

@data_fields = split(":",$AF::data); // Separate the fields in our data segment
$home = $data_fields[0]; // The first field is the initial host

if ( $home eq $hostname ) {
        // If we have been sent back to the original host,
        // alert the administrator, because something is amiss!
        print stderr "A system has been hacked!\n";
    return;
}

// Are any of these malicious programs running?
SWITCH: {
        if (/ircd/) { $compromised = 1; last SWITCH; }
        if (/DOS-master/) { $compromised = 1; last SWITCH; }
        if (/hackedprogram/) { $compromised = 1; last SWITCH; }
        $compromised = 0;
}

// Uh oh, one of the malicious programs were running.
// Get a snapshot of the running process and migrate home.
if ( $compromised ) {
        $AF::data .= ":" . $hostname . ":" . $process_list;
        AF::Move($home)
}

@itinerary = split(",",$data_fields[1]);  // Retrieve our itinerary
$to = $itinerary[rand $#itinerary+1]; // Select a random host to migrate to
AF::Move($to); // This system is clean, move to the next host
```

Figure 4.1: Intrusion Detection Agent

injection, the agent data segment contains the following information:

```
injector.nrl.csci.unt.edu:  <--- (Injector Host)
alpha.ameslab.gov,          <--- (Itinerary)
beta.nrl.csci.unt.edu,
gamma.csci.unt.edu,
delta.unt.edu
```

In general, an agent would contain more detailed data. For instance, it might reference the data segment for known signatures of malicious programs. However, we have reduced the data size for brevity.

The purpose of this agent is simple, detect and report when a host is executing one or more malicious programs. For this example, the agent assumes that the *ps*, and *hostname* executables have not been replaced with trojaned versions. More specifically, the programs return correct data when they are executed.

When this agent migrates to a host, it begins by retrieving a list of running processes on the host. In addition, it records the name of the host that it resides on. In order to perform its task, the agent retrieves data from its data segment, via the global *AF::data* variable. Since the agent is written to understand the format of its data segment, it knows how to parse the AF::data variable in order to retrieve information.

Once the agent determines the name of the host on which it currently resides, the agent retrieves the name of the host that initially injected it. For this example, the agent migrates between the hosts in its itinerary until it has found a compromised host. Once it has found a host that fits the compromise criteria, the agent returns to the host from which it was injected. Thus, as the agent code illustrates, if the current host is identical to the injector host, then the agent has found a compromised host and has returned home to alert the administrator.

When the agent migrates between hosts in its itinerary, it engages in a series of checks to determine whether a system that it currently resides on has been compromised. This involves a regular expression match that searches for particular strings within a process listing (i.e., *ps*). If it finds a match, the agent flags a variable to indicate that the host has been compromised. If a compromised host is found, the agent records some basic information about the host (i.e., process listing and hostname) and returns to the injector to report the status. Otherwise, the agent selects a random host from its itinerary and migrates, repeating the process forever.

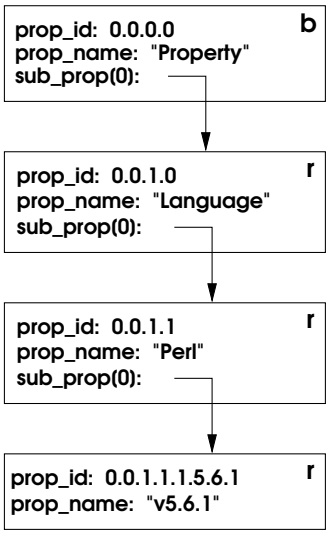
Thus far, we have described the goals and actions that the agent will take to complete its task. If the agent was static, its execution would be simple; however, our agent must be able to migrate randomly between four hosts found in its itinerary. This means that at some point, the agent will have to engage in the transfer protocol.

For this particular example, the agent uses *platform migration*, where functionality to engage in the transfer protocol is provided by the library loaded by the module.

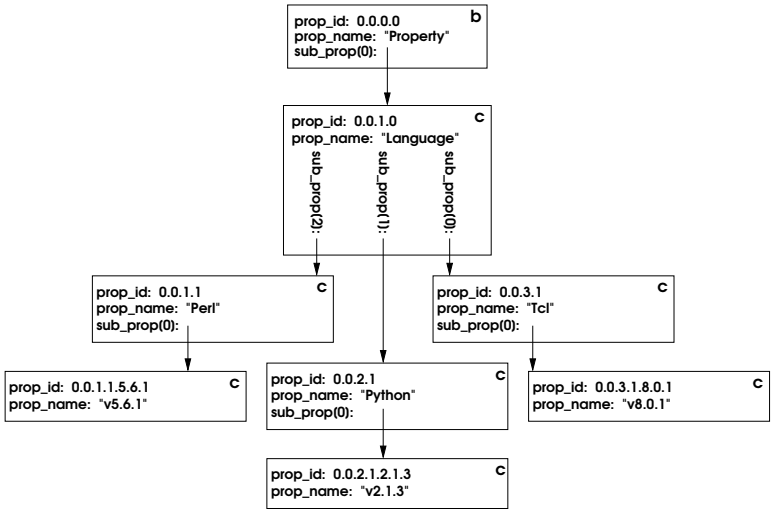
In order for the transfer protocol to work, the agent and the hosts listed in its itinerary must provide property hierarchies that describe their capabilities and requirements. Illustrated in Figure 4.2a, the agent's property hierarchy is rooted with the standard *Property* property which contains a sub-property path that describes its only requirement, a version 5.6.1 Perl interpreter.

Similar to other data structures, a property hierarchy can easily be represented using a textual string. This greatly simplifies transmission since it allows the agent to carry its description as a text string. In its current implementation, the textual format uses the same fields that were introduced with the property data structure discussed earlier. Thus, to represent the agent's property hierarchy, the textual string appears as follows (newlines added for clarity):

```
(b 0.0.0.0 Property
   (r 0.0.1.0 Language
      (r 0.0.1.1 Perl
         (r 0.0.1.1.1.5.6.1 v5.6.1)
      )
   )
```



(a) Agent Properties



(b) Host Properties

Figure 4.2: Property Hierarchies

)

To make this example interesting, suppose that in addition to a Perl module, each of the hosts have also loaded two other language modules, namely Python and Tcl. Thus, after the module loading protocol has occurred, each host should contain a property hierarchy similar to the one displayed in Figure 4.2b. Similar to the agent, the host also maintains a textual representation of its property hierarchy for transmission.

At this point, our environment consists of an agent and five hosts that are executing the DADS daemon. Each host has loaded the Perl, Python, and Tcl modules and each are ready to accept incoming agents. Therefore, to begin agent execution, we use the special injector program to build the agent and boot-strap it into the DADS infrastructure. In particular, our injector program:

1. Reads the agent code.
2. Reads the agent data.
3. Reads the agent property description.
4. Connects to a remote DADS host.
5. Engages in the transfer protocol.

As soon as the agent has been successfully injected, the agent is forwarded to a module for execution. The agent is now fully autonomous, thus, the agent should not return to the injector host until it has discovered that a malicious program is running on one of the hosts in its itinerary.

An agent must engage in the transfer protocol to migrate to a remote host. For the example agent, the transfer protocol informs a remote DADS that an incoming agent would like to use the Perl interpreter. To accomplish this, both the agent and host use their property hierarchy information to make decisions about the services that are provided. This requires the exchange of property information, which is used to initialize an environment for agent authentication (not used in this example) and execution.

To better understand the transfer protocol, assume that our agent is to be injected from the host, *injector.nrl.csci.unt.edu*, to a remote DADS-enabled host, *alpha.ameslab.gov*. Therefore, to inject the agent, we execute the injection program which engages in an information exchange similar to what is displayed in Figure 4.3. For this agent, the exchange occurs in four steps. The first step transmits the entire agent to the remote host. The complete agent is transmitted since many forms of authentication (i.e., hashing, PKI, etc.) may require a complete image of the agent. Discussed earlier, the host cannot trust that an agent is sending correct

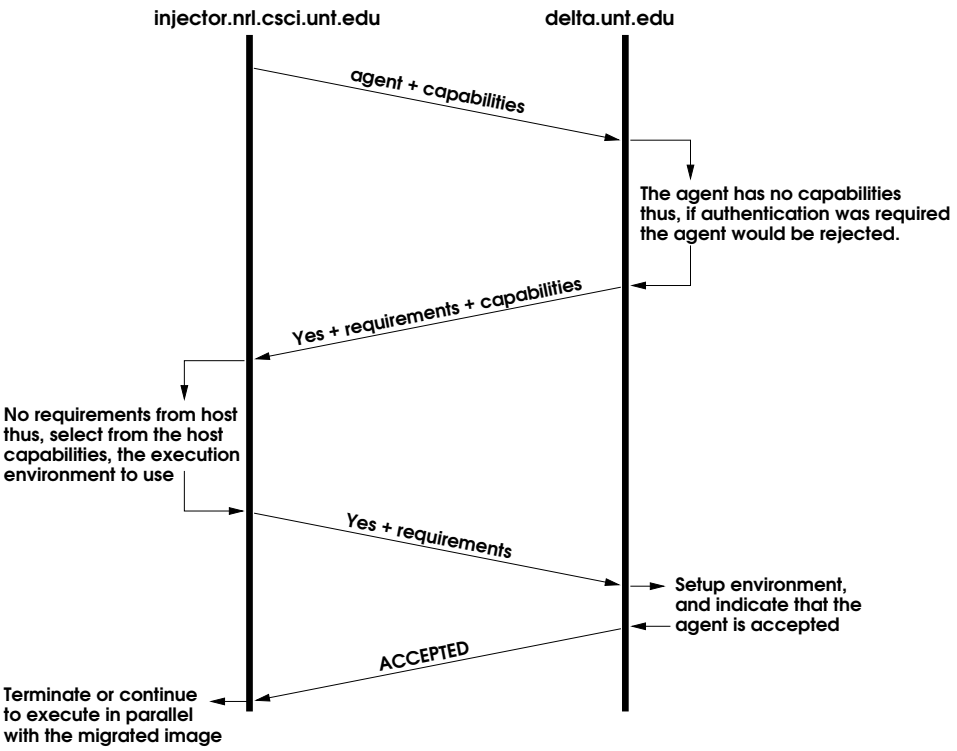


Figure 4.3: Transfer Protocol Exchange

authentication information, thus, a local copy is kept for verification. It is important to note that this initial copy of the agent is the only copy that is sent during the entire protocol. If the protocol results in agent acceptance, then the copy is sent to a module for further execution and the remote agent is informed that the agent was accepted. However, if at any point during the protocol the agent is not accepted (i.e., fails authentication, host cannot support agent), then this copy is deleted and the remote agent is informed that migration was denied.

In addition to a copy of the agent, the first step of the protocol dictates that the agent send a property hierarchy comprised of the agent's capabilities. As Figure 4.2a illustrates, the example agent has no capabilities, only requirements. When the host receives this information, it selects from the agent's capabilities the services that it wants the agent to use. If the host determines that the agent is not capable of supporting a service which the host requires, the host denies the migration request. In this example, the host does not have any requirements (see Figure 4.2b), hence, the protocol moves to the next step.

Next, the host sends a reply containing two separate property hierarchies. The first hierarchy describes the capabilities that were selected by the host for the agent to use (in this case none). The second hierarchy contains the host's capabilities. Similar to how the host selected capabilities that it wanted the agent to use, the agent now

uses this second property hierarchy to select the services it wants to use on the host. If the agent determines that the host cannot support the services it requires, the connection is terminated and the protocol is reset. In this example, the agent's only requirement is the Perl interpreter, which is supported by the host. Therefore, the protocol moves to the third step where it replies with the selections that it made.

Finally, as soon as both sides of the protocol have agreed that they can support each other, the host replies with ACCEPTED. This lets the remote agent know that the copy (already local to the DADS) is being forwarded to a module for execution. In general, when an ACCEPTED reply is received, the agent that initiated the migration has two options. The first option is to terminate, allowing the recently migrated code to continue the agent's legacy. The second option is to continue execution, thereby allowing the migrated code to execute in parallel with itself.

There are several issues that we ignore in this example. First, if a language module crashes during agent execution, the agent is lost. Second, there are no mechanisms for security, hence, neither a host nor agent can be fully trusted. While they are both important issues, each of these problems could be solved as described earlier with the fault tolerance and security modules.

CHAPTER 5

SUMMARY

The application of an agent-based solution requires a system that provides delivery and execution services for mobile code and data. In addition, such a system should be efficient and secure. There are many agent systems that answer this demand, however, many of them are based in virtual machine architectures as well as proprietary designs. Not only has this impeded the advancement of the agent paradigm, but it has also created a library of agent systems that use fairly complex methods which are incompatible with other systems. While standards such as FIPA and MASIF have been developed to address issues such as system interoperability, many systems still do not conform to them.

To address some of these issues, this thesis has presented the design and implementation of a Distributed Agent Delivery System (DADS). Designed for simplicity and flexibility, the DADS is an agent platform focused on the delivery and execution of multi-lingual agents. Using a modular design similar to AgentTcl, the DADS uses its module system to act as a gateway to resources. In addition to language execution, the DADS module system potentially provides access to an unlimited range of

services.

This thesis has discussed some of the advantages and disadvantages found in many of todays mobile agent systems. This included a review of several infrastructures, ranging from those that use a strictly virtual machine design (i.e., Telescript, Aglets) to systems designed for multi-lingual agents (i.e., AgentTcl). Each of these systems offer new concepts that have influenced the DADS design in one form or another.

In addition, this thesis has discussed the issues that are encountered when designing an agent for a mobile agent infrastructure. As described above, agent systems commonly use an agent design that has been developed to fit the needs of their own, sometimes proprietary, infrastructure. More specifically, the agent is defined to fit a set of particular requirements (i.e., language, security, etc.) which optimize properties of that system. Commonly, this limits the portability and interoperability of the system when it is used in conjunction with other agent systems.

In contrast to a proprietary design, the DADS uses an agent that exists solely as a container for code mobility. Strongly influenced by the agents of TACOMA, a DADS agent relies on a three segment model which is used to store code, data, and a set of descriptive properties. Without limiting an agent's content, code and data segments store arbitrary sequences of bits. This means that DADS agents are inherently multi-lingual, hence, they can use any language and format for their data.

However, it is imperative for this type of free-form agent to understand its capabilities and requirements. Thus, DADS agents also contain a specially designed property segment to maintain a concise description of the agent's code and data segments.

A DADS daemon addresses the requirements of a host by allowing it to participate in agent based solutions. In particular, the DADS satisfies a host's obligation to accept and execute mobile agents from a network. Since a host ultimately executes an agent using its resources, it is imperative for that host to provide an entry point. As a result, the DADS uses its daemon to listen on the network for incoming agents. When an agent has been accepted, it is forwarded by the daemon to its collection of loadable service modules. In the DADS, modules are responsible for providing agent services.

Influenced heavily by AgentTcl, modules allow the DADS to support a heterogeneous blend of agents. Supporting services ranging from language execution and security to fault-tolerance, modules provide agents with access to resources. In addition, this thesis discussed the design and implementation of a generic module, which can be used as a model for the design of future modules. In general, modules provide the system administrator(s) with a customizable agent system that can be adjusted to fit the policies of an institution.

As a binding mechanism between the free-form DADS agent and the customizable

DADS daemon, the DADS system uses a special transfer protocol to facilitate agent migration across heterogenous systems. More specifically, it defines the procedure used to exchange information from an agent's property segment, which allows each entity to decide whether it can support the services of the other.

Agents are autonomous entities. Further, if an agent creates another agent, through a process referred to as *cloning*, then it is possible to have a very large number of agents in a network. If left unchecked, this could potentially lead to an excessive population causing degradation in network performance. In general, agent population control is an important topic to consider in an agent system. However, since it is a fairly new topic, there are many new ideas that have been proposed to solve this problem. In particular, a naming system could be used where new agents are registered with a central naming authority [22]. Thus, before a new agent is created, the naming authority could be queried facilitating a decision whether too many agents already exist. Second, a biological model based on pheromones could also be used [2]. Thus, as agents move from host to host, they leave a virtual residue that decays over time. As new agents migrate to this host, they use the residue to determine the last time an agent visited that host. In general, this will enable an agent to make a more informed decisions about cloning. Regardless of its implementation, experiments in population control require an agent platform that can emulate new

concepts without sweeping redesigns of the supporting infrastructure. In the DADS, concepts like this can be added using new modules, making it a very attractive system for experimental agent research.

Future Work

Conforming to the strategies discussed throughout this thesis, the implementation of a stable version of the DADS system is almost complete. Having developed the foundation code for the DADS daemon, future development is now focused on the creation of a library of service modules. Since they provide the core agent services, the development of additional modules will hopefully increase the viability and usefulness of the DADS system. In particular, it is imperative that the initial focus for module development be placed on mechanisms for security and fault-tolerance. Currently, work is being done to adapt the security model used for Grid computing [11] using *proxy-certificates* [24] into a module based agent-authentication mechanism. Further, development has already begun on modules that support the Python, Perl, and C language.

Future development will address the design of tools that can be used for the management and visualization of agents executing within the DADS system. In particular, tools that can graphically monitor the status of DADS-enabled hosts and

report the identities of resident agents would benefit both administrators and users alike.

As soon as a stable module base has been established, DADS development will move into a performance analysis phase. This phase will provide a review of the DADS as it stands in comparison to other agent systems. Since it provides a generic platform for agent execution, it is expected that the DADS will perform slower when compared to systems such as AgentTcl. In particular, the transfer protocol will have a significant effect on the amount of time required by an agent migration. Since the AgentTcl system knows the exact format of the agents that it receives, it will not need to engage in an agreement protocol like that of the DADS. Consequently, this creates another area for future work, optimizing the transfer protocol. In addition, work will be done to migrate certains aspects of the transfer protocol to use a standards based format, such as the eXtended Markup Language (XML).

In its current implementation, the DADS has been developed using the libraries and functionality provided by the Linux[1] operating system. In the future, the DADS will be ported to other operating systems, making it cross-platform.

The DADS is an extendable agent system designed for flexibility. As such, it provides a solid base for future work in agent research.

[1]Linux is a trademark of Linus Torvalds

APPENDIX A

NETWORKING OBJECTS

```
//////////////////////////////////////////////////////////////////
//
//      The following is a declaration of the Network class.
//      This class is designed to encapsulate all information
//      related to networking.  It provides a super class from
//      which Client and Server objects will derive methods.
//
//////////////////////////////////////////////////////////////////

#ifndef __NETWORK_H__
#define __NETWORK_H__

class Network {
        public:
                Network();
                ~Network();
                struct hostent *GetHostByName(char *hostname);
                int GetServiceByName(char *servname);
                int GetSocketNumber();
                int Read(char *buf,int buflen);
                int Write(char *buf,int buflen);
                int SetSockOpt(int optname, const void *optval,
                               socklen_t len);
                int SetNonBlock();
                void SetSocketNumber(int sd);
                void SetTimeout(int seconds);
                int IsTimedOut();

        protected:
                int timeout;
                int sockdesc;
                struct sockaddr_in sockinfo;
                timeval last_activity;
};

#endif __NETWORK_H__
```

```
//////////////////////////////////////////////////////////////////
//
//      The following is a declaration of the Server class
//      This class is designed to encapsulate all information
//      related to a tcp/ip server.  This class is derived from
//      the Network class which contains a wide variety of network
//      functionality not specifically restricted to a server.
//
//////////////////////////////////////////////////////////////////


#ifndef __SERVER_H__
#define __SERVER_H__

#include "Network.h"
#include "Client.h"

class Server : public Network {
        public:
                Server(int domain, int type, int proto, int port);
                ~Server();
                void Bind();
                void Listen();
                Client *Accept();
                void Close();
                void Shutdown(int how);

        protected:
};

#endif __SERVER_H__
```

```
////////////////////////////////////////////////////////////////
//
//      The following is the declaration of the Client class
//      This class is designed to encapsulate all information
//      related to a tcp/ip client.  This class is derived from
//      the Network class which contains a wide variety of network
//      functionality not specifically restricted to a client.
//
////////////////////////////////////////////////////////////////


#ifndef __CLIENT_H__
#define __CLIENT_H__

#include "Network.h"

class Client : public Network {
        public:
                Client(int cs, const sockaddr_in *addr,
                       socklen_t len);
                Client(int domain, int type, int proto, char *host,
                       int port);
                ~Client();
                void Connect();
                void Close();
                void Shutdown(int how);

        protected:
                socklen_t socklen;
};

#endif __CLIENT_H__
```

```
//////////////////////////////////////////////////////////////////
//
//      The following is a declaration of the TCPClient and
//      TCPServer class.  These classes are designed to
//      encapsulate all information related to TCP specific
//      client and servers.  These classes are derived directly
//      from the Client and Server classes.
//
//////////////////////////////////////////////////////////////////


#ifndef __TCP_H__
#define __TCP_H__

#include "Client.h"
#include "Server.h"

class TCPClient : public Client {
        public:
                TCPClient(char *host, int port);
                ~TCPClient();
};

class TCPServer : public Server {
        public:
                TCPServer(int port);
                ~TCPServer();
};

#endif __TCP_H__
```

```
//////////////////////////////////////////////////////////////////
//
//      The following is a declaration of the UDPClient and
//      UDPServer class. These classes are designed to
//      encapsulate all information related to UDP specific
//      client and servers.  These classes are derived directly
//      from the Client and Server classes.
//
//////////////////////////////////////////////////////////////////


#ifndef __UDP_H__
#define __UDP_H__

#include "Client.h"
#include "Server.h"

class UDPClient : public Client {
        public:
                UDPClient(char *host, int port);
                ~UDPClient();
};

class UDPServer : public Server {
        public:
                UDPServer(int port);
                ~UDPServer();
};

#endif __UDP_H__
```

BIBLIOGRAPHY

[1] Aglets Web site: http://www.trl.ibm.co.jp/aglets/index_e.htm, 2002.

[2] K. Amin and A. Mikler, *Dynamic Agent Population in Agent-Based Distance Vector Routing*, ISDA2002: Second International Workshop on Intelligent Systems Design and Applications, Atlanta, USA, August 2002.

[3] K. Amin, J. Mayes, and A. Mikler, *Agent-based Distance Vector Routing*, Proceedings of the Third International Workshop, MATA 2001, August 2001.

[4] Joachim Baumann, Fritz Hohl, and Kurt Rothermel, *Mole - Concepts of a Mobile Agent System*, Technischer Bericht TR 1997/15, Fakultat Informatik, Universitat Stuttgart, 1997.

[5] C. Baumer, M. Breugst, S. Choy, and T. Magedanz, *Grasshopper - A Universal Agent Platform Based on Omg Masif and Fipa Standards*, http://citeseer.nj.nec.com/440987.html, 2002.

[6] J.D Case, M. Fedor, M.L. Schoffstall, and C. Davin, *Simple Network Management Protocol (SNMP). RFC 1098.*, April 1989.

[7] Crystaliz Inc., General Magic Inc., GMD Fokus, International Business Machine Corporation, *Mobile Agent Facility Specification*, June 1997.

[8] Gianpaolo Cugola, Carlo Ghezzi, Gian Pietro Picco, and Giovanni Vigna, *Analyzing Mobile Code Languages*, Mobile Object Systems: Towards the Programmable Internet, Springer-Verlag: Heidelberg, Germany, pages 93–110, 1997

[9] William M. Farmer, Joshua D. Guttman, and Vipin Swarup, *Security for Mobile Agents: Authentication and State Appraisal*, Proceedings of the Fourth European Symposium on Research in Computer Security, Rome, Italy, pages 118–130, 1996.

[10] FIPA Mission Statement: http://www.fipa.org/about/mission.html, 2002.

[11] Ian Foster, *The Anatomy of the Grid: Enabling Scalable Virtual Organizations*, Lecture Notes, Computer Science, vol. 2150, 2001.

[12] Robert S. Gray, *Agent Tcl: A Flexible and Secure Mobile-Agent System*, Proceedings of the Fourth Annual Tcl/Tk Workshop (TCL 96), Monterey, California, 1996.

[13] R. Gray, D. Kotz, G. Cybenko, and D. Rus, *D'Agents: Security in a Multiple Language Mobile-agent System*, Mobile Agents and Security, Lecture Notes in Computer Science, No. 1419, pages 154-187, Springer-Verlag, 1998.

[14] Internet Assigned Numbers Authority, IANA, http://www.iana.org, 2002.

[15] W. Jansen and T. Karygiannis, *NIST Special Publication 800-19 - Mobile Agent Security*, Jansen W. Karygiannis, T.: NIST Special Publication 800-19 - Mobile Agent Security. National Institute of Standards and Technology, 2000.

[16] Dag Johansen, Robbert van Renesse, and Fred B. Schneider, *Operating System Support for Mobile Agents*, Proceedings of the 5th Workshop on Hot Topics in Operating Systems, pages 42–45, May 1995.

[17] Neeran M. Karnik and Anand R. Tripathi, *Design Issues in Mobile-Agent Programming Systems*, IEEE Concurrency, Vol. 6, No. 3, pages 52–61, 1998.

[18] D. Lange and M. Oshima, *Programming and Deploying Java^{TM} Mobile Agents with Aglets^{TM}*, Addison Wesley Longman, Reading, MA, 1998.

[19] Kre J. Lauvset, Dag Johansen, and Keith Marzullo, *TOS: A Kernel of a Distributed Systems Management System*, Lauvset K. J., Johansen D., Marzullo K. (2000) TOS: A Kernel of a Distributed Systems Management System, 2000.

[20] D. Milojicic, W. LaForge, and D. Chauhan, *Mobile Objects and Agents (MOA)*, Proceedings of USENIX COOTS'98, Santa Fe, 1998.

[21] Object Management Group, OMG, http://www.omg.org, 2002.

[22] Karim Taha and Thomi Pilioura, *Agent Naming and Locating: Impact On Agent Design*, D. Tsichritzis, 1999.

[23] Tomas Sander and Christian F. Tschudin, *Protecting Mobile Agents Against Malicious Hosts*, Lecture Notes in Computer Science, Vol. 1419, page 44, 1998.

[24] S.Tuecke and D. Engert and Ian Foster and M. Thompson and L.Pearlman and C.Kesselman, *Internet X.509 Public Key Infrastructure Proxy Certificate Profile*, Internet Draft: draft-ggf-gsi-proxy-03.PDF, 2002.

[25] W. Richard Stevens, *Unix Network Programming - Networking APIs: Sockets and XTI*, Vol. 1, 2nd edition, pages 397–424, 1998.

[26] J. E. White, *Afterword: White, J.E., Telescript Restrospective*, Mobility: Processes, Computers, and Agents, Addison Wesley/ACM Press, Dejan S. Milojicic and Frederick Douglis and Richard Wheeler, p. 493, 1999.

[27] J. E. White, *Telescript Technology: Mobile Agents*, Software Agents, AAAI/MIT Press, 1996.