UC San Diego
UC San Diego Electronic Theses and Dissertations

Title
Behavior-based remote executing agents

Permalink
https://escholarship.org/uc/item/33q311kf

Author
Hung, Eugene

Publication Date
2006

Peer reviewed|Thesis/dissertation

eScholarship.org Powered by the California Digital Library
University of California

UNIVERSITY OF CALIFORNIA, SAN DIEGO

Behavior-Based Remote Executing Agents

A dissertation submitted in partial satisfaction of the
requirements for the degree Doctor of Philosophy
in
Computer Science

by

Eugene Hung

Committee in charge:

Professor Joseph C. Pasquale, Chair
Professor William G. Griswold
Professor Bill Lin
Professor Ramesh R. Rao
Professor Bennet S. Yee

2006

The dissertation of Eugene Hung is approved, and it is acceptable in quality and form for publication on microfilm:

____

____

____

____

____

Chair

University of California, San Diego

2006

TABLE OF CONTENTS

	Signature Page	iii
	Table of Contents	iv
	List of Figures	vii
	List of Tables	viii
	Acknowledgements	ix
	Vita, Publications, and Fields of Study	x
	Abstract	xi
I	Introduction	1
	A. Previous Solutions	2
	B. Our Solution: reAgents	4
II	System Environment Model	8
	A. System Environment Model	8
	1. Definitions and Assumptions	8
	2. Regions	10
	B. Location of Customizing Logic	12
	C. Advantages of Client-Based Code Mobility	14
III	Related Work	16
	A. Active Networks	16
	B. Dynamic Proxies	17
	C. Publish / Subscribe	18
	D. Client-based Customizers	19
	E. Mobile Agents	20
	F. Design Patterns	20
	G. Summary	21
IV	ReAgents	22
	A. Design Specification	22
	B. ReAgent Architecture	25
	1. Client/ReAgent/Server Linkage	25
	2. Components	26
	C. Relation to Model	28
	D. Summary	28

V General Behaviors 29
A. Filter 30
B. Monitor 32
C. Cacher 35
D. Collator 37
E. Summary 40

VI ReAgent API 41
A. The `ReAgent` Interface 41
B. The Behavior Interface 43
1. Customizing Logic Interfaces 43
C. The Converter Interface 47
D. The Protocol Interface 47
E. Summary 48

VII Programming Examples 50
A. Examples of Usage 50
1. Example 1: Basic Data Filtering reAgent 51
2. Example 2: Customizing Communications 55
3. Example 3: Integration into Traditional Applications 56
4. Example 4: Changing the CL to Filter Ads 57
5. Example 5: Using Other Behaviors 59
6. Example 6: Using Multiple Behaviors 60
B. Best Practices 62
1. Determining Location 62
2. Determining Protocols 62
3. Determining Behaviors 63
C. Summary 63

VIII Internal API 64
A. The AgentSystemHandler Interface 64
B. The ReagentHandle Interface 66
C. The ReagentHost Interface 66
D. The ReAgent Protocol 67
E. Summary 67

IX ReAgent Implementation 68
A. Language: Java 68
B. Mobile Code System: JAE 68
1. The ExtensionHandler class 69
2. The JAEHandle class 69
3. The JAEHost class 72
4. The ReagentExtension class 72
C. Standard Protocols 74

1. Plain-text Protocol Implementation 74
2. HTTP Implementation 76
D. Summary 81

X Experiments 82
A. Experiment 1: Scalable Communications 82
B. Experiment 2: Effectiveness of reAgent Paradigm 84
1. Experiment 2a: Response Time Comparison 85
2. Experiment 2b: Application Performance Comparison 87
C. Experiment 3: Deployability Overhead 88
D. Summary 92

XI Conclusion 93

Bibliography 95

LIST OF FIGURES

I.1 Client-ReAgent-Server Model 5

II.1 Operative network environment 9
II.2 Server view of environment 11
II.3 Client view of environment 11
II.4 Bounding of Client Region 13

IV.1 ReAgent Architecture 26

V.1 The Filter Behavior 30
V.2 The Monitor Behavior 32
V.3 The Cacher Behavior 35
V.4 The Collator Behavior 38

X.1 Bandwidth Regulator Output 84
X.2 Stock Traders Setup 87
X.3 End-to-end comparison times 90
X.4 Overhead of filtering (log-based) 91

LIST OF TABLES

X.1 Comparison of Response Time for Various Paradigms 86
X.2 Mobile ReAgent vs. Stationary ReAgent 88

ACKNOWLEDGEMENTS

This dissertation, in part, is a reprint of material as it appears in WWW 2004 (Hung, Eugene; Pasquale, Joseph. "Web Customization Using Behavior-Based Remote Executing Agents"), ASWN2004 (Hung, Eugene; Pasquale, Joseph. "Using Behavior Templates to Design Remotely Executing Agents for Wireless Clients"), and of material that has been submitted for publication in AAMAS06 (Hung, Eugene; Pasquale, Joseph. "ReAgents: Behavior-based Remote Agents and Their Performance"). The dissertation author was the primary researcher and author of these papers, and the co-author listed in this these publications directed and supervised the research which forms the basis for this dissertation.

As for personal acknowledgements, I'd like to dedicate this dissertation to my advisor, Dr. Joseph Pasquale, without whose efforts it would not have eventually been written, and my committee, which was very patient and accomodating with my progress. I'd also like to thank my parents, Wilfred and Ya-Yi Hung, and my sister and brother-in-law, Valerie and Thomas Leung, for continually encouraging me. Finally, this work would not be complete without mentioning some of the many friends I've made at UCSD whose company stimulated me to conduct the research described within: Michael Davis, Dan Harting, Eron Jokipii, Jason Lee, Vivian Lin, Jeremy Martin, Travis & Tina Newhouse, Cameron & Katie Parker, Sriram Ramabhadran, Vik Srimurthy, and Jesse Steinberg.

VITA

1995	B.S., University of California, Berkeley
1995–1996	MICRO Fellowship, University of California, San Diego
1996–1997	National Semiconductor Fellowship, University of California, San Diego
1998	M.S., University of California, San Diego
1998–2003	Research Assistant, University of California, San Diego
2006	Ph.D., University of California, San Diego

PUBLICATIONS

"Improving Wireless Access to the Internet by Extending the Client/ Server Model." Proc. European Wireless Conference, Florence, Italy, February 2002.

"Web Customization Using Behavior-Based Remote Executing Agents", Proc. of the 13th Int'l World Wide Web Conference (WWW 2004), New York, NY, May 2004.

"Using Behavior Templates to Design Remotely Executing Agents for Wireless Clients", 4th Workshop on Applications and Services in Wireless Networks (ASWN 2004), Boston, MA, August 2004.

"ReAgents: Behavior-based Remote Agents and Their Performance", 5th Int'l Conference on Autonomous Agents and Multi-Agent Systems (AAMAS 06), Hakodate, Japan, May 2006. (to appear)

ABSTRACT OF THE DISSERTATION

Behavior-Based
Remote Executing Agents

by

Eugene Hung
Doctor of Philosophy in Computer Science
University of California, San Diego, 2006
Professor Joseph C. Pasquale, Chair

ReAgents are remotely executing agents that customize Internet applications for thin/weak clients. A reAgent is essentially a "one-shot" mobile agent that acts as an extension of a client, dynamically launched by the client to run on its behalf at a remote, more advantageous, location. ReAgents simplify the use of mobile agent technology by transparently handling data migration and run-time network communications, and provide a general interface for programmers to more easily implement their application-specific customizing logic. This is made possible by the identification of useful remote *behaviors*, i.e., common patterns of actions that exploit the ability to process and communicate remotely. Examples of such behaviors are filters, monitors, cachers, and collators. In this dissertation, we identify and analyze a set of useful reAgent behaviors, describe how to program and use reAgents, and show that reAgents provide a scalable, deployable, and effective solution to the problem of client heterogeneity.

Chapter I

Introduction

The trend towards smaller, wireless Internet-access devices has led to complications in Internet client/server-based application design. Servers must now be designed to handle a broad range of computing power and/or connectivity quality. An increasingly frequent scenario is that of transient, sub-standard clients popping into the network unexpectedly, demanding unusual services, and finding the services unsatisfactory due to the server's inability to flexibly deal with the shortcomings of the client device.

To make these problems concrete, consider the typical client/server-based electronic-commerce application of a user purchasing merchandise over the Internet. If the user's client device has below-average computing resources or network connectivity, or if the user has unusual demands, problems can arise. A client device with limited network bandwidth and a black-and-white display might be too slow when the server sends extraneous images and videos of the merchandise. Another device with an unreliable connection to the Internet may be unable to verify that a transaction was completed, possibly sending a duplicate transaction order due to an intervening disconnection. A third device, due to the susceptible nature of its wireless connection, may require levels of security beyond the server's ability to supply. These types of problems degrade quality of service, and will increase in frequency as client devices grow more diverse in their needs and resources.

Our approach to addressing these types of problems is to support the remote processing of client/server application requests and (especially) responses via dynamically deployed, remote agents, which carry out work tailored to the particulars of the client device. The remote agent acts as an intermediary between client and server, but maintains the client/server model (by operating as a server to the original client and as a client to the original server). So, an application running on a client device with a small, limited display would benefit from a remote agent operating at or near the server that filters the rich server data into a spartan, plain-text response. On a client device with an unreliable connection, application performance would be improved from a remote agent operating at the boundaries of the problematic portion of the connection (not necessarily the end-points) that stabilizes it with a disconnection-aware protocol. Finally, for users with unusual demands, a specialized service could be implemented as part of the remote agent, acting as a higher-level service relative to the services provided by the server.

I.A Previous Solutions

Active Networks and Server-based Solutions

The idea of enhancing client applications with remote code is not new, but previous efforts have been divided on how and where to provide this functionality. The active networks approach [41] is to place the logic "inside the network.", having it run on network-central machines such as routers. Another common approach is to attempt to have servers adapt to the specifics of each individual client. Server-provided CGI scripts/forms, and the group of technologies bundled into the Open Mobile Alliance (OMA) [31] are two examples of this type of approach, where the server programmer is responsible for anticipating common client problems and catering to them.

These types of approaches can have deployability or scalability problems. Placing the remote code in the network is difficult in terms of deployment, can

also negatively impact other network applications, and is open to violating the end-to-end principle[34]. Server-based approaches, although easy to deploy on an individual level, lack generality: some servers may support a specific client, but other servers may not. Even if one restricted communications only to servers that catered to one's needs, performance is unsatisfactory if the client environment changes in an unanticipated manner or new clients with different, unusual needs arrive.

Intermediaries

A more scalable solution is to have the remote code operate as a user-level intermediary on a machine between the client and server. Such an intermediary would act as a standard client as viewed by the server by communicating with it using the pre-established client/server protocol. The intermediary would also act as a specialized server for the client, with the ability to, for example, filter data received from the server into a more suitable form for the client.

A popular type of intermediary, for which there is much research and experience, is a proxy that provides a static service, usually pre-installed by an administrator, to which a client sends its requests for processing before it gets passed on to the server. Proxies are a good solution for customizing large groups of clients with similar demands. For example, all clients connecting via low-bandwidth links to a higher-speed network might use a filtering proxy that operates beyond these links. However, static proxies are limited in scope and typically inflexible in where they can be located. If a client needs special customizing logic that operates optimally at a specific location (such as at or near the base station for a wireless client), it may be difficult to install such a special proxy at that location. Furthermore, proxies installed by parties other than the client suffer from similar scalability problems that arise from server-based customization.

At the other extreme of types of intermediaries is the *mobile agent*[12]. By a mobile agent, we simply mean a unit of code that is capable of migrating

from the client to a remote site, acting on behalf of the client. Mobile agents may migrate in a *weak* fashion (where only the agent's data and code, but not its execution state, are moved autonomously) or in a *strong* fashion (weak migration plus migration of execution state). As [26] demonstrates that a weak-migration agent is equivalent to a strong-migration agent, we include both types of migration in this discussion of mobile agents.

Mobile agents, with their potential to migrate during execution, are extremely flexible and powerful customization tools. And unlike server-based solutions, they scale well with increasing client heterogeneity as each different client can use its own type of agent to alleviate its problems. The downside of mobile agents is that they typically require specialized agent systems to handle the semantics and security problems that are a byproduct of code migration, even weak migration. They are also not easy to program, as programmers are generally not familiar with the mobile code programming paradigm.

I.B Our Solution: reAgents

Given these extremes, we seek a middle-ground solution, with the following goals:

- provide the programmer a better way to handle its needs and limitations
- be transparent to servers (i.e., do not require modifications to servers)
- be easy to program and use

To meet these goals, we propose a remote code mechanism that is, simply put, more flexible than proxies but less complicated than fully-general mobile agents. We achieve this compromise by, first, adopting a form of "one-shot" mobile agents, simply called a *reAgent* (for "remotely executing agent"). Unlike a general mobile agent, which can move to multiple machines during its computation and

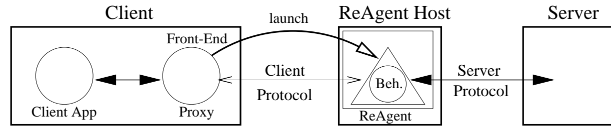

Figure I.1: Client-ReAgent-Server Model

retain its state and identity, a reAgent moves exactly once: before it begins execution. The reAgent cannot move after it has begun execution, or even launch other reAgents (only the user may launch reAgents). By restricting its movement in this fashion, we can restrict, and thus, make guarantees about its potential behavior. This model is similar to that of remote evaluation [38].

A reAgent is launched to operate at a remote location that is superior in, for example, available computing or network resources. This location is known as the *reAgent host* (Fig. I.1). The reAgent then acts as a customized proxy for that particular user, passing through server requests and processing the response before forwarding it to the client. This allows for extending the capabilities of the client in a customized fashion.

By moving its own provided code to a better location, the client gains extra power to better deal with its limitations and needs. And, the client can represent itself to the server as a standard client via its reAgent, thereby keeping the server code unchanged.

Several advantages arise from limiting movement to one hop. By avoiding some of the security issues introduced by code that can roam from site to site, infrastructural support is simplified. Also, with a stationary remote agent acting on its behalf, the client gains the benefits from remote execution without adjusting its client/server architecture: the reAgent acts as a server by taking requests and returning responses. Finally, technical problems associated with maintaining and updating program state during migration are avoided, without losing much functionality, a view supported by [24].

The most novel aspect of our approach is that reAgent code is strictly derived from a template library of *behaviors*. Behaviors are useful patterns of processing and communication that are the result of restricting and simplifying the form of movement of reAgents. Not only do behaviors capture common useful forms of client/agent/server interactions, but importantly, can be *specialized* for particular application needs via code parameters. The simplest and most common example is that of a "filter" behavior, which in general processes a response from a server before passing it on to the client, and can be specialized in terms of how the server data will be reduced. A client device with a black/white display would benefit from a filter that strips the color from image files, while another client device with a tiny color display would benefit from a filter that shrinks the images to a manageable pixel count. The filter behavior is a general behavior, while the specific type of filter is a specialized behavior.

So far, the useful general behaviors we have identified are :

- *filtering*, by reducing the form of a server response before it is communicated to the client, to reduce the overhead over low-bandwidth links, or to reduce client storage and processing
- *monitoring*, to improve application reaction times to critical changes in state at the server by observing and triggering actions closer to the server
- *caching*, by saving commonly-accessed data at a location close to the client to improve responsiveness when there is high network latency between the client and the server and the client does not have sufficient system resources to efficiently operate a local cache
- *collating*, by moving the distribution point of a request, copies of which will be forwarded to numerous servers, to a more efficient location where the responses can then be fused into a single result

These behaviors can be used not only in isolation, but also in combination. Thus, a single reAgent is composed of one or more behaviors (each of which

is specialized for its originating client). By identifying the general behaviors of reAgents and using them as building blocks for development, we provide a simple, structured way of building and deploying reAgents that efficiently extend clients in an application-specific, scalable fashion.

The rest of this dissertation is organized as follows.

Chapter II covers the construction of a model of the operative network environment, to fully outline the problem situation. Chapter III reviews previous related work in the area of Internet application customization to identify the pros and cons of previous approaches. Then, the main contribution of this dissertation follows: Chapter IV describes the concept of reAgents and explains why their usage leads to scalable, deployable, and effective customization of network applications. Chapter V describes specific components used for building reAgents: behaviors, converters, and protocols. Chapter VI describes the external ReAgent API, with Chapter VII illustrating the use of this API by providing a series of progressive examples. Chapter VIII provides a detailed description of the internal implementation for launching and supporting reAgents. Then, Chapter IX discusses the details of our implementation of this internal interface on top of a bare-bones mobile code system, Java Active Extensions. Experiments showcasing the versatility, performance, and usability of reAgents are described in Chapter X. Finally, Chapter XI concludes this dissertation and proposes directions for future work.

This chapter, in part, is a reprint of material as it appears in the Thirteenth International World Wide Web Conference (WWW2004) under the title "Web Customization Using Behavior-Based Remote Executing Agents" and as it appears in the Fourth Workshop on Applications and Services in Wireless Networks (ASWN2004) under the title "Using Behavior Templates to Design Remotely Executing Agents for Wireless Clients". The dissertation author was the primary researcher and author of these papers, and the co-author Joseph Pasquale directed and supervised the research which forms the basis for this chapter.

Chapter II

System Environment Model

In this chapter, we present a model of a system environment where an advanced remote code mechanism would be useful. If a user is not running a client application in an environment which fits this model, the conclusions drawn by this model, and consequently this dissertation, may not apply. However, we have designed the model so that it applies to many common Internet applications. Meanwhile, the model facilitates explaining the rationale behind our approach: application-based remote processing.

II.A System Environment Model

II.A.1 Definitions and Assumptions

We define the network environment as a traditional client/server environment, with the network conceptually divided into two segments, *the typical segment* T and the *atypical segment* A (Fig. II.1).

T consists of physical network link(s) that are fairly homogeneous, predictable, and stable (i.e., the Internet in general). A, which represents the client device's connection to the Internet, contains links less predictable in their attributes and possibly significantly different from most links in T.

Each network segment is a conceptual unification of its individual com-

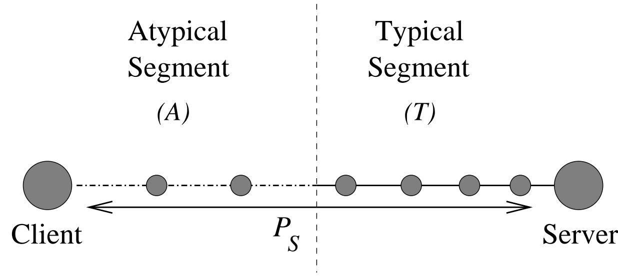

Figure II.1: Operative network environment

posite links, and has gross characteristics of performance and reliability that can be abstracted from the finer-grained characteristics of its individual links. For example, on a link-level basis, the reliability of a segment is only as strong as its weakest physical link. When there is only one route between the client and server, the bandwidth of a segment cannot exceed the least bandwidth of its physical links. Finally, the latency of a segment is at least the sum of the latencies of its physical links.

Within this environment, we make the following assumptions:

- The server, when communicating with a client, assumes clients to be fairly powerful devices that communicate with the server through links similar to those in the typical segment.

- If a client device deviates from this expectation, the server is not expected to handle it.

- The server expects the client to communicate with a pre-defined protocol P_S, which we refer to as the *server protocol.*

- The client, when communicating with a server, knows its network limitations and how it deviates from the server's model of a standard, powerful, well-connected client.

- The client also understands the above server protocol P_S.

As stated earlier, our following argument is based on these assumptions holding. However, the lack of one assumption does not necessarily render the argument invalid: if, for example, there exist servers capable of catering to a client device's specialized needs, that merely means the problem scenario does not exist for that specific client device at that specific server.

II.A.2 Regions

A *region* is an encapsulation of environmental awareness that incorporates some of the above assumptions into our model. All machines in a region (and the code that runs on those machines) possess knowledge of any deviations from the standard Internet environment. For simplicity, deviations are assumed to be static. In particular, a wireless network is only considered atypical if its operation is distinguishable from that of a wired network.

In addition, machines outside of a region lack any knowledge of any environmental deviations that operate within that region. Note that unlike a firewall, a region does not preclude the ability to communicate with machines in different regions — it merely expresses the boundary of environmental awareness.

Our model defines two regions: the client region and the server region. The client region consists of the client device and the atypical segment, to model the assumption that the client user understands its deviations. Meanwhile, the server region consists of the server machine(s) and the typical network segments that connect them to the atypical segment. This models the server's lack of knowledge about any client problems. (For purposes of simplicity and clarity, we limit the subsequent discussion to a server region containing a single server, but multiple servers are also supported by this model.)

Code in the server region operates as if the network environment were as depicted in Fig. II.2. This view consists of the exterior of the client region, $U - C$ (which contains the server region S), and the physical network links between C and S that make up the client-server network connection (which is categorized

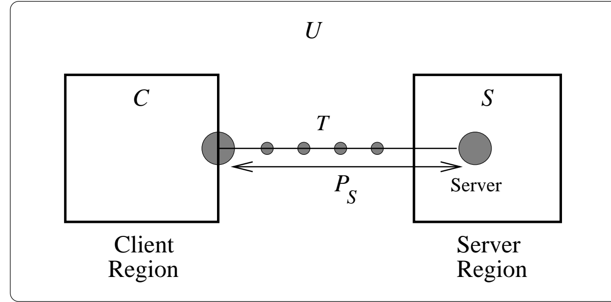

Figure II.2: Server view of environment

as a typical network segment). Thus, server code is only required to assume that there is a client, operating on a machine remote to the server, that connects and makes requests to the server using the server protocol P_S. It does not know about the atypical network segment at all.

In contrast (Fig. II.3), code operating in the client region operates in an environment consisting of the exterior of the server region $U - S$ (which contains the client region C). The code developer is not required to understand either the implementation of the server architecture or the details of the typical segment, but only the server protocol P_S and the point of communication where the server can communicate with the client using P_S.

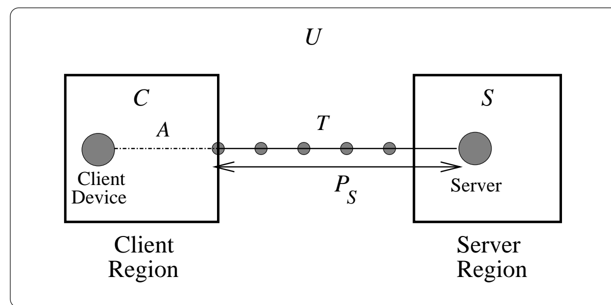

Figure II.3: Client view of environment

With the definition of regions, we can now model the problem of client heterogeneity. We describe the client region C' as different from a typical, standard client region C in one or more attributes. Examples of differences between C' and C include:

- Physical limitations of client device (small display, inadequate memory)
- Atypically low bandwidth in connection to Internet
- Atypically unreliable communications
- Atypically insecure communications
- Excessive latency between client and server

All of these differences are encapsulated within the client region, modeling the server's lack of information about the client's special problems.

II.B Location of Customizing Logic

Given this model, and the concept of using remote code for customizing application performance, it is important to decide where the customizing logic should be placed. There are three possible areas where such customizing logic is based. They are:

- The network
- The server region
- The client region

As briefly alluded to in the Introduction (and discussed in more detail in Chapter III), each of these three locations has been used in previous solutions for customizing performance. If the customizing logic is placed within the network, both client and server will have to deal with this change in their world view, making it hard to deploy. Also, the overhead of the mechanisms supporting the customizing logic is potentially incurred by all applications, regardless of whether they use them or not, thus lowering efficiency.

Meanwhile, if the customization logic is based at the server, then each change in a client causes a change in every server. This solution does not scale

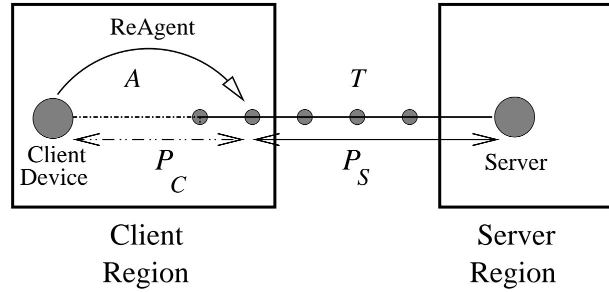

Figure II.4: Bounding of Client Region

well: as the heterogeneity of client devices (represented by the addition of different client regions) increases, support required in the server universe also increases, as each server must add customization logic to handle each new client region.

Thus, we have chosen to base the location of the customizing logic in the client region. The customizing logic is originated by the client and placed on a machine that resides in the typical segment, beyond the atypical segment. Once the logic is placed, we bound the client region to have the client device at one end and the customizing logic at the other end, so that the atypical segment is masked from the server. (Fig. II.4)

From the model, we can see that the customizing logic, being in the client region, is exposed to the problems of the client region, but that the server is not. Instead, the server treats the customizing logic as the actual client, as they are both part of the opaque client region. Thus, the server need not change when new client problems arise, promoting deployability and scalability. Meanwhile, the client is still able to communicate with the server by providing the customizing logic with an understanding of the server protocol P_S. Moreover, with the addition of client-originated customizing logic as an intermediary between client and server, the client can now communicate with said logic using a custom *client protocol* P_C. Such a protocol could be used to ameliorate communication problems over the atypical segment, or provide the client with run-time control over the customizing logic.

II.C Advantages of Client-Based Code Mobility

Given this model, customization within the client region stands out as having no intrinsic scalability or deployability problems. Yet previously well-researched client-based customization solutions such as mobile agents are not widely in use.

The reason is not because mobile agents are inapplicable to the problem of client heterogeneity. Any solution that supports code mobility and remote operation is powerful enough to solve individual client problems. For example :

- Resource-poor client devices can have computation take place on the server hosting the agent (the agent host), bypassing the client environment as much as possible.
- Bandwidth use over the atypical segment can be reduced by filtering data at the agent host to a smaller amount before sending it over the atypical segment, or having the agent be responsible for sending out messages to multiple servers from its remote position.
- If the client and server are using a connection-less, unreliable protocol such as UDP, then the reAgent can add reliability on top of the protocol by re-sending lost data. Or, in the case of an extreme environment such as an interplanetary network [10], the reAgent might improve reliability by implementing forward error-correction.
- Insecure links can be strengthened by having the agent encode transmissions (with a client-specific algorithm as the customizing logic) before sending to the client for decoding.
- Latency can be reduced by moving computation closer to the object the agent is communicating with, or by bypassing unnecessary communication with the server by storing a copy of previously retrieved data with the agent (e.g., a cache).

The examples above show that many applications could benefit from the advantages provided by mobile agents. Why, then, given such widespread application, have mobile agents not been successful? Before describing our answer and solution, we first revisit past solutions to the problem in more detail in the next chapter.

Chapter III

Related Work

The customization of applications for improved performance has seen a variety of past research solutions. Active networks, dynamic proxies, publish/subscribe messaging, and mobile agents are but a few of the solutions advanced to solve this problem. In this chapter, we describe the past research in this area in detail and explain how their solutions do not lend themselves to scalable, deployable, and effective customization.

III.A Active Networks

The active networks approach [41] argues for putting customizing logic at the network level with the use of programmable packets, or "capsules", that can change the behavior of the network. While active networks are able to effectively support a wide variety of client devices, they do so at the expense of deployability. Using a network-based approach such as active networks makes severe demands on the Internet infrastructure, going against the end-to-end principle[34] that is arguably the primary reason behind its success. Violating the end-to-end principle means that every application, including those that do not need customizing logic, must deal with the overhead and side effects of active networks. In addition, active networks also raise serious security concerns due to the possibility of packets reprogramming the network.

Some active networks have been implemented by researchers, the most prominent being ANTS [44], NetScript [36], and SwitchWare [2]. In each of these projects, the central problem of deploying active networks into the existing infrastructure remains. The ALAN project [14], which moves the active network functionality into the application-level, addresses the issue of flexible deployment, but does not explain a general, structured method for building applications, the subject of this dissertation.

III.B Dynamic Proxies

Another customization solution lies in the use of proxies, which act as intermediaries between client and server. Proxies are different from our approach in that they are not necessarily mobile (movable from site to site), and thus tend to be part of the existing infrastructure rather than originating from the client in response to a certain problem. Such infrastructure-based approaches work well with a general class of clients/problems, but are inadequate for unusual or unexpected problems.

While traditional proxy applications concentrate on caching Web results for improved performance and for controlling Internet access through firewalls [27], some have focused on actively customizing application behavior (dynamic proxies). In [8], the idea of an Active Cache, or a dynamic proxy that acts to help improve caching for dynamic Web objects, was first proposed. In Active Cache, content providers provide specialized code in the form of a cache applet that intermediate caching servers execute to produce a new version of the cached object. More recently, [28] describes a large-scale, server-based framework for caching dynamic Web content and facilitating personalized services, called WebGraph. WebGraph is designed to be deployed without client-side support, so it is primarily targeted at groups of clients instead of individual clients.

Server-based customization techniques such as dynamic proxies are highly

deployable because they do not require changing the underlying network infrastructure and represent the most popular approach of solving the problem of client heterogeneity. However, such techniques, while effective in a piecemeal fashion server-by-server, do not fit our goal for a global, scalable, server-independent customization solution. While such a goal seems ambitious, it is actually a necessary response to the continual creation of new client devices with correspondingly different demands. Such a process makes it difficult, if not impossible to anticipate every potential variation in client capabilities. And even if such variations were able to be anticipated completely, not every server has the resources to handle every potential variation. Finally, even if such resources were available, there is no guarantee that the service the user wants to use would make use of them to accommodate the user. The user may have no recourse in a server-based customization scenario, being completely dependent on the server for a solution to its particular problem.

The Active Names project [42] describes the use of a dynamic proxy, introduced by either server or client, that customizes how resources on a wide-area network are located and transported to a client. While this provides dynamic customization, it does not satisfy the goal of completely avoiding server participation. For scalability purposes, the client must introduce the customizing logic.

III.C Publish / Subscribe

Content-based publish/subscribe (a.k.a. pub/sub) is another technology area that can be applied towards the goal of scalable customization. Potential clients can *subscribe* to an event feed tied to a data server. When the server *publishes* new data to the feed, all clients are notified that they need to update their feed. The feed acts as an intermediary that allows clients and servers to be added and removed independently of each other. The feed can also operate filters to ensure that clients get customized traffic, thus supporting scalable, deployable

customization. Past research in this area includes pub/sub systems such as Siena [9], Gryphon [6], Solar[11], and Fulcrum [7].

While pub/sub achieves scalable, deployable customization, and is capable of emulating certain subsets of reAgent behaviors such as filtering newsfeeds for news stories of particular interest, most pub/sub systems do not have the power to support the full range of reAgent behaviors. A reAgent supports the execution of general customized code with restricted inputs and outputs, so a reAgent provides greater programming generality than a pub/sub system.

III.D Client-based Customizers

In order to be less dependent on the server, researchers have developed customizers with more client-side support. Reference [46] describes the implementation of a client-proxy-server framework that supports the on-demand downloading of custom filters (the customizing logic) to a proxy. The proxy then executes the filter on communications from the server before passing it onto the client. Unlike our work, this framework focuses on filtering applications instead of all types of applications that could benefit from mobile code.

A more flexible Web-oriented customization scheme is detailed in [40], which describes the implementation of a middleware architecture that supports adaptive Web-based proxies called Customizers. Customizers are deployed on behalf of a client, and are split into two points of control, so as to separate the individual extension of a Web browser from its remote, location-dependent computation. However, it is optimized for use over an HTTP client/server connection and not a more generic client/server connection.

An industry-based approach to individualized customization lies in IBM's Web Intermediary suite[5]. Here, the customizer (called a plugin) is an HTTP request processor, and can be placed either between client and server as a proxy, or act as a HTTP server of its own. A Java development kit is provided for poten-

tial plugin developers, allowing for rapid deployment of custom plugins through a provided library of monitors, editors, and (document) generators. However, this work is targeted for HTTP connections only, and it does not use mobile code to take advantage of deployment to optimal locations for customization.

III.E Mobile Agents

A significant area of past research in client-based customization has been based upon *mobile agents*. Mobile agents are pieces of customizing logic that have a persistent identity, moving around the network to multiple sites. The IBM Aglets Workbench [25] and the D'Agents project [17], from industry and academia respectively, are prominent examples of systems that support the execution of mobile agents. A fuller description of these and other important agent systems, as well as the current state of mobile agent research can be found in [18].

Mobile agents provide a robust solution for addressing the problems of client heterogeneity: they are both deployable and scalable. However, despite having several years for the idea to incubate, mobile-agent-based applications are rare. This is not due to lack of theoretical value: [20], [19], and [43] describe applications which take advantage of mobile agents. But, value notwithstanding, few applications based on mobile agents are in widespread use. Most application programmers are either unaware of the paradigm of mobile agents, or uninterested in handling the details necessary to support client-specific desires. Thus, our work differs from previous mobile agent literature by concentrating on a method that reduces the complexity of building agent-based applications.

III.F Design Patterns

[45] describes the problems facing the development and deployment of mobile agent (and mobile code) applications. Consequently, there has been some work attempting to extend the idea of design patterns to mobile code [15]. Design

patterns are a software engineering approach for facilitating the development of object-oriented applications. While design patterns have previously been proposed for use in designing mobile agent applications [3, 35, 37], these patterns differ from our approach, in that they tend to be at a higher level of abstraction instead of application-oriented. For example, all reAgents would conform to what might be called a One-Hop design pattern, as all of the reAgents move only one network hop. In this work, each of these One-Hop reAgents are differentiated by the application behavior and underlying support protocols.

III.G Summary

The problem of providing customizing logic to client applications is not new, but previous efforts have been divided on how and where to provide this functionality. Network-based solutions such as active networks face severe deployment issues. Server-based solutions such as dynamic proxies do not scale well in handling unexpected client problems or non-standard protocols. Finally, previous client-based solutions such as mobile agents are promising, but not widely used. The root of their lack of usage lies not in their applicability, but in their complexity and unfamiliarity. What is needed is a client-based remote processing technology more similar to client/server than mobile agents (to enhance usability) while providing scalable and deployable benefits to the problem of client heterogeneity. The next chapter will describe the design and architecture of our solution to this problem: reAgents.

This chapter, in part, is a reprint of material as it appears in the the Fourth Workshop on Applications and Services in Wireless Networks (ASWN2004) under the title "Using Behavior Templates to Design Remotely Executing Agents for Wireless Clients". The dissertation author was the primary researcher and author of this paper, and the co-author Joseph Pasquale directed and supervised the research which forms the basis for this chapter.

Chapter IV

ReAgents

As can be seen from the previous chapters, client-launched intermediaries for remote customization is not a new idea, but previous solutions were complex and hard to deploy. A *reAgent* is a client-originated intermediary with restrictions placed upon its movements and its behavior (its actions). Such a behavior acts as a template for building a reAgent, defining a specific sequence of communication and processing.

In this chapter, we explore the design goals and top-level architecture of reAgents.

IV.A Design Specification

The design goal of reAgents is to provide a scalable, deployable, and effective solution to the problem of client heterogeneity. To that end, reAgents are designed with the following key features:

Scalability through Client-Based Deployment Per the model in chapter II, the client side may experience problems unknown to the server side. Thus, the solution must come from the client side, not the server side, in order to properly scale as new clients (and their corresponding problems) are introduced into the environment. A reAgent is therefore launched by a client device to an intermediate

site, the *reAgent host*, to be executed in a client-specific fashion.

One-shot Movement ReAgents use a restricted form of mobile agent technology by limiting movement to one migration only. This type of movement is named *one-shot* to emphasize that reAgents move once and then terminate at that location, in contrast to the unlimited movement capabilities present in most mobile agents. Most of the advantages of client-based code mobility, as described in Section II.C, do not depend upon the ability to move multiple times. Thus, by simplifying the reAgent movement potential to one network hop, we have the ability to create effective pieces of mobile code that are easy to deploy.

General Behaviors For most environment problems that are a result of client heterogeneity, a general solution can be found that treats that problem. For example:

- Excessive latency between computation and target is solved by moving the computation closer to the object, or bypassing communication if unnecessary through caching.
- Limited bandwidth can be solved by having the reAgent filter data at the reAgent host before it is sent back.
- Insecure links can be neutralized by having the reAgent encrypt transmissions before sending to client for decryption.
- Traffic can be shaped to a specific pattern by having the reAgent manage the traffic flow.
- Unreliable links can be overcome with the reAgent re-sending lost data.

These solutions can be encapsulated as a *general behavior*. A general behavior describes common patterns of action for a reAgent (specifically, its movement, communication, and logical processing) that takes advantage of remote processing to improve the application's performance. In this manner, we guarantee

that reAgents constructed from useful general behaviors will provide useful, effective customization.

Behavior Templates Part of the problem facing the widespread use of mobile agents lies in their implementation and development. Most network application programmers are used to programming in the client/server model, or lately, the peer-to-peer model. In contrast, the mobile agent model adds several dimensions (such as code movement, remote execution semantics, and intermediary communication handling) that are unfamiliar to most programmers. By categorizing reAgents in terms of general behaviors, we can write code to handle these unfamiliar areas for each general behavior, and use it as a template for creating specific reAgents that behave in that general manner.

For example, data filtering is a common, useful reAgent behavior. A filtering template is provided to handle the movement and internal logic for a general filtering reAgent. All a developer needs to do is to insert a client-specific data-filtering algorithm (such as stripping color, or shrinking the image size), and a custom reAgent will be created. Using a behavior template, one can easily create custom, fully-functional reAgents as specific instances of a general intermediary behavior.

Summary With these key design features, we conclude that reAgents are :

- Scalable — they are originated by the client
- Effective — applications that use them gain a strong advantage from remote execution (the advantages provided by their behaviors).
- Deployable — built with a simple, reusable, integrated approach that handles the interface between movement, communications, and the custom logic

This is in accordance with our design goals. It now remains to fill in the details of this design specification and show its implementation.

IV.B ReAgent Architecture

IV.B.1 Client/ReAgent/Server Linkage

ReAgents are targeted towards client/server environments, and are especially useful where servers (and clients) are not easily amenable to change. Given a potential client/server interaction, the client and server must somehow be linked to the reAgent, which is interposed in their communications. The reAgent communicates with the server as a traditional client: it sends the server a request and receives a response. From the server's point of view, the reAgent is just another standard client. However, the reAgent needs to intercept the client request to a server, without necessarily changing the client application code. This is accomplished by using a *front-end proxy* that runs on or near the client device. The client application's output is redirected to the front-end proxy's input for forwarding to the reAgent. The front-end proxy can do this because it is responsible for launching the reAgent to the reAgent host (using the reAgent API).

Once the reAgent is launched, avenues of communication are automatically set up between the front-end proxy and the reAgent, as well as between the front-end proxy and client application (the client application's output is redirected to the front-end proxy's input). Then, the proxy acts as a client to the reAgent by sending requests (from the client application) to the reAgent, and acts a server to the client by passing responses from the reAgent to the client application. In this manner, the implementation of the client application and server remains unchanged.

For the reAgent code to migrate for execution on the reAgent host, some middleware system that supports remote execution must be available. Our implementation of reAgents is not tied down to a specific format of remote execution, which would limit its scope. Instead, we leverage existing work in mobile code systems by translating the launch procedure into the appropriate code movement calls for each system. These movement calls are translated at run-time when the

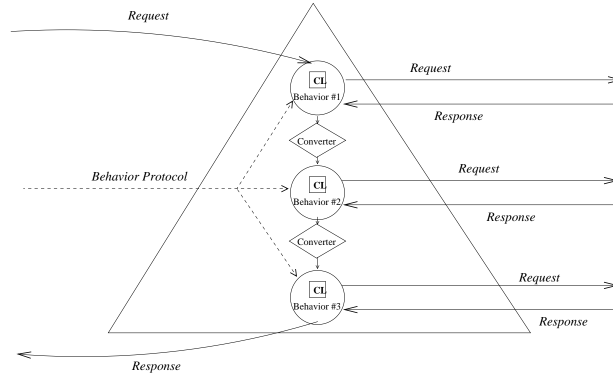

Figure IV.1: ReAgent Architecture

agent is launched, with the type of middleware on the target reAgent host is specified in a configuration file. (Note that the underlying mobile code middleware system is completely transparent to the programmer, who must only understand the simplified reAgent interface.) As explained in IX, we use a locally-developed mobile-code middleware system [30].

IV.B.2 Components

Behaviors A reAgent is composed of specialized behaviors, which in turn are instantiations of a general behavior. As a programming object, a general behavior is equivalent to a template, consisting of base logic (the *BL*) that captures the general actions of the behavior, and an API that allows it to be specialized with programmer-provided custom logic (the *CL*) and to be controlled by the user during run-time (control methods).

A behavior operates in the following fashion: as part of the reAgent, it waits until it receives a request. This request is passed as an input to the BL, which may call some methods implemented by the client (the CL). At some point during the BL, request(s) are made to a server (specified as a parameter when the reAgent is configured with the behavior), which returns a response. This response is also passed through the BL (and may call another method of the CL) before being output. At no point does the behavior initiate communication with the client;

the reAgent containing the behavior handles all the input and output. During execution, the client can tell the reAgent to invoke the control methods of the behavior, which allow control of the reAgent's behavior from an external point.

Converters Because input and output are handled at a higher level in a consistent fashion, multiple behaviors can be combined in a chain. The behaviors are chained together through the use of *converters*, i.e., code that can convert the response output of one behavior into the request input for another one. Then, the higher-level reAgent code can run a behavior whose response output is fed to the converter, and then get the converter's output of a client request to feed to the next behavior in the chain.

Protocols A reAgent is able to customize its communication using protocol components. Each behavior communicates with the server using the *server protocol* (the protocol recognized by the server, and like the server, specified as a parameter at reAgent configuration time) and with the client using a *client protocol*. The client protocol is customizable by the client and allows the client to substitute a protocol that is better suited than the server protocol for the connection between client device and reAgent host. For example, a useful scenario for a custom client protocol arises when a portion of the network path near the client is relatively unreliable, e.g., wireless access. One could launch a reAgent to a location beyond the unreliable portion, e.g., at or beyond the wireless base station, set up a more stable, reliable protocol between the client and reAgent, while continuing to use the standard server protocol between the reAgent and the server.

Finally, the client can explicitly control the reAgent during run-time using the *reAgent protocol*, which is at a higher level than the client/reAgent and reAgent/server protocols discussed above. The reAgent protocol is fixed (it is defined by the behaviors of the reAgent), and is used by the client to invoke a behavior's control methods.

IV.C Relation to Model

With this design and architecture, reAgents comfortably fit within the system environment model described in Chapter II. ReAgents are created and launched by a user who understands how the client region deviates from the server region, and knows how the reAgent can customize the server data to alleviate these problems. The reAgent location designates the boundary between the client and server regions, so that the client protocol is used to communicate within the client region and the server protocol is used to communicate within the server region.

The reAgent host is picked by the user to divide the network into typical and atypical segments. The route from the reAgent host to the server must be a typical network segment (and by definition, can use the server protocol without problems), while the route from the reAgent host to the client must be atypical (and uses the client protocol to address any network problems). To achieve this condition, the reAgent must migrate to a host that is closer to the server than any bottleneck on the atypical segment.

IV.D Summary

In this chapter, we described the design specification for reAgents. We introduced the reAgent components of behaviors, converters, and protocols, and showed how they interact. Finally, we tied our design back to the system environment model. In the next chapter, we will describe and analyze a library of general behaviors that we use to create templates for reAgent categorization and construction.

Chapter V

General Behaviors

We developed the definition of behaviors based on their usefulness in the design of reAgents. A behavior is "useful" if it exhibits benefits that are derived from a reAgent's ability to operate remotely. These benefits come from some combination of, but not limited to, the following:

- use of remote computational resources
- avoiding or minimizing the effects of a problematic portion of the network (high delay, low bandwidth, low reliability, etc.)
- ability to act autonomously on behalf of the client in a customized fashion

The following sections catalog the useful behaviors we have identified. For each behavior, we present a description of the behavior, an outline of its base logic, an example of its use, and an analysis of its performance. To clarify the exposition, we show the base logic in pseudo-code. For consistency, we limit the usage discussions to Web applications, as they are good illustrations of beneficiaries of using reAgents, and were the focus of our first developed reAgents [21].

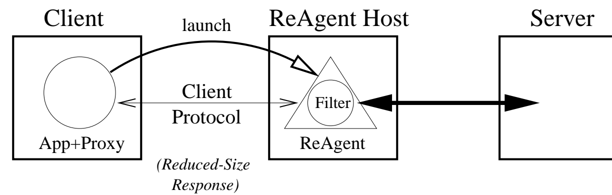

Figure V.1: The Filter Behavior

V.A Filter

Description The Filter behavior (Fig. V.1) is used whenever a server response needs to be reduced (in size) before reaching the client. The reAgent interposes itself between the client and server, filters the server responses into a smaller, possibly more suitable format for the client, and sends this filtered result back to the client. The CL (customizing logic) is the application-specific algorithm that defines *how* to reduce the data.

Base Logic

```
input: request

---------------------------------------------------------------
serverProtocol.send(request)
response = serverProtocol.receive()
newResponse = filterLogic.filter(response, args)
---------------------------------------------------------------

output: newResponse
```

The reAgent passes the request through to the server, runs the filtering algorithm on the server response, and sends the new response back to the client.

Application The Filter behavior is designed for scenarios where the server data is too large for the client. A common scenario involves browsers on clients with limited capabilities, such as small battery-powered wireless devices (e.g., PDAs). General features of such a device include limited network bandwidth as well as low-fidelity rendering of data, so filtering by removing extraneous or unusable data before sending it to the browser would reduce required bandwidth and reduce delay without significantly impacting the perceived quality of the data.

Analysis The following is a performance analysis of the Filter behavior. In this analysis and those that follow, we assume that network latency (i.e., propagation time) and network message processing overhead are negligible (relative to the other factors we consider). All overheads that are attributable to the the reAgent itself, e.g., its launching time, are encapsulated as a single variable (λ).

One way a filter is effective is if it reduces end-to-end server-to-client delay. In straight client/server, the delay (D_{cs}) to return a file of size $\mathcal{S}$ is the transmission time given by file size divided by client-server bandwidth (B_{cs}):

$$D_{cs} = \frac{\mathcal{S}}{B_{cs}}$$

The delay as a result of using reAgents is equal to the the transmission time between the the server and reAgent ($\frac{\mathcal{S}}{B_{rs}}$) plus the processing time for the filter ($\mathcal{P}_{filter}$), plus the overhead of using the reAgent (λ), plus the transmission time between the reAgent and the client ($\frac{\alpha\mathcal{S}}{B_{cr}}$), where α is the percentage of original data left over from the filter:

$$D_{crs} = \frac{\mathcal{S}}{B_{rs}} + \mathcal{P}_{filter} + \lambda + \frac{\alpha\mathcal{S}}{B_{cr}}$$

To compare D_{cs} and D_{crs}, we calculate the *speedup*, which represents the percentage improvement of the reAgent approach. When the speedup is positive, reAgents are superior; when the speedup is negative, traditional client/server implementations are superior.

The speedup is derived from the following equation:

$$speedup = 1 - \frac{D_{crs}}{D_{cs}} \tag{V.1}$$

Plugging in the values of D_{crs} and D_{cs} into the speedup equation above, and setting B_{cr} equal to B_{cs} (assuming that the bandwidth of a set of links is equal to the bandwidth of the slowest individual link) results in a speedup of

$$speedup = \frac{(1-\alpha)\mathcal{S}/B_{cs} - (\mathcal{S}/B_{rs}) - \mathcal{P}_{filter} - \lambda}{D_{cs}}$$

If we express the ratio of the bandwidths as $\rho = \frac{B_{cs}}{B_{rs}}$, then *speedup* > 0 when

$$(1 - \alpha - \rho)\frac{\mathcal{S}}{B_{cs}} - \mathcal{P}_{filter} \quad > \quad \lambda$$

Thus, reAgents become more advantageous as

- α decreases (the filter reduces more data)
- ρ decreases (the bandwidth ratio of the client/reAgent to reAgent/server network segments becomes much smaller)
- $\mathcal{P}_{filter}$ decreases (the filtering algorithm processes more quickly)
- B_{cs} decreases (the overall bandwidth is small)
- $\mathcal{S}$ increases (there is more data to filter)
- λ decreases (overhead of using reAgent decreases)

So if client-reAgent bandwidth drops, or original data size gets larger, the filtering reAgent becomes more effective.

V.B Monitor

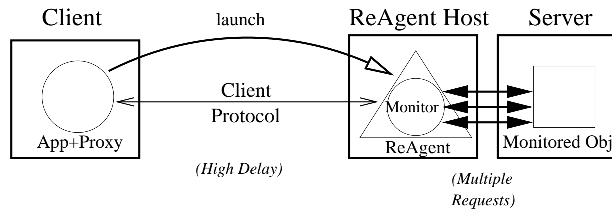

Figure V.2: The Monitor Behavior

Description The Monitor behavior (Fig. V.2) is designed for use in applications that have a need to frequently examine the state of a remote object (on a far-away server) until a certain state is observed. The calculation of the next monitor attempt, plus the response evaluation function, forms the CL.

A reAgent that uses the monitor behavior is placed on a site close to the object that is being monitored, for the purpose of reducing the time of receiving a critical state change and sending the trigger action to the server. This is important for applications that require a real-time response to sudden changes in environment, such as a stock ticker or online bidding auction.

Base Logic

```
input: request, requestParam

---------------------------------------------
do

    /* pause before checking */
    queryTime =
      monitorLogic.calcnextQuery(requestParam)

    sleep (queryTime - currentTime)

    /* check remote object */
    serverProtocol.send(request)
    response = serverProtocol.receive()

while (monitorLogic.testResponse(response) -> FALSE)
----------------------------------------------

output: response
```

The behavior repeatedly calculates the next time to query the server, queries the server at that time, and then checks to see if a trigger state has been reached. Once the trigger state is reached, monitoring is terminated and the response that triggered the state change is returned.

Application A simple example of an application for a Monitor involves intelligent auto-refresh of a Web browser. Many pages auto-refresh at fixed intervals. A Monitor can bypass the automatic refresh and refresh at its own customized rate. This can be advantageous when the Monitor is sensitive to network conditions

and adjusts the rate depending on the amount of traffic. More importantly, the intelligent auto-refresh only updates the client when the server changes, and will not force a refresh when the data remains unchanged, saving bandwidth.

Analysis To evaluate the monitor, we assume that the monitor will send a message to the server n times before the trigger condition is satisfied.

Under straight client/server, the total delay between sending a request to the server and processing its reply is

$$T_{cs} = n \times D_{cs}$$

where $n =$ the number of queries before the trigger condition is satisfied.

With a reAgent, the total delay is

$$T_{crs} = D_{cr} + nD_{rs} + \lambda$$

as the delay between client and reAgent is only paid once. Equation V.1 gives a reAgent speedup of

$$speedup = 1 - \frac{D_{cr} + nD_{rs} + \lambda}{nD_{cs}}$$

Solving for $speedup > 0$, we find that reAgents are better for monitoring when

$$(n-1)D_{cr} > \lambda$$

Thus, reAgents become more advantageous as

- n increases (more queries before trigger state is reached)
- D_{cr} increases (more delay between client and reAgent that is avoided)
- λ decreases (overhead of using reAgent decreases)

Therefore, many queries, or high delay between the client and the reAgent, points to using a monitor reAgent to gain a performance advantage.

V.C Cacher

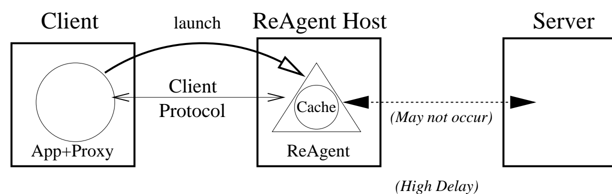

Figure V.3: The Cacher Behavior

Description The Cacher behavior (Fig. V.3) is used for storing recently retrieved server data at a nearby location with the expectation that it will be accessed again, thus improving future performance. When previously retrieved data is requested again, the nearby stored copy is retrieved instead of the distant original. The cache replacement policy forms the CL. This behavior is especially useful for applications that have frequent yet identical requests to remote servers, such as occurs in Web browsing.

Base Logic

```
input: request

------------------------------------
key = cacherLogic.hash(request)

if (cacherLogic.lookup(key) -> TRUE)
  response = cacherLogic.get(key)
else
  serverProtocol.send(query)
  response = serverProtocol.receive()
  cacherLogic.replace(key, response)
------------------------------------

output: response
```

This behavior uses customized logic on the request input to decide whether or not to pass along the request to the server. If the request has not been made recently, the behavior generates a “key” which gets associated with the request, and then uses the request to contact the server. When the server responds, the behavior associates the data in the response to the key of the request and stores both

items in a database, i.e., the cache, before outputting the response data. When a request is made that matches a key in the cache, the behavior will bypass sending the request to the server and immediately return the associated cache data.

The behavior is in charge of inserting, removing, and retrieving data contained within the cache. Insertion of data into the cache happens whenever the server sends the behavior a response. Cached data and its corresponding key are removed whenever the amount of storage allocated to the cache begins to run out, or by special order of the client. Data is retrieved from the cache when the client request key matches a key within the cache. While the behavior defines these general actions, particulars regarding cache policy (such as which cache entries to replace first when the cache is full) are supplied as part of the CL.

Application Caching of frequently accessed Web pages is so beneficial to performance that most major Web browsers support some form of caching. With no intermediate hosts, the server data is stored on the client device. While storing the cached data on the client device is optimal for minimizing network delay, some client devices have such small amounts of memory that cache performance is seriously degraded by running locally. These resource-poor clients would benefit greatly from moving the cache from the client device to a nearby location with sufficient resources.

In such situations, a reAgent with a Cacher type of behavior can be created, along with a client-specified cache replacement policy and size, and launched to the nearby machine to effectively operate a cache for the browser.

Analysis To evaluate the cacher in both scenarios, end-to-end delays are compared.

Under straight client/server, the delay between sending a request to the server and processing its reply is simply D_{cs}.

Under a caching reAgent, the delay between sending a request to the server and receiving its reply depends on whether the item requested is in the

cache. A delay between client and reAgent is always incurred. When the item is not found in the cache, a delay is also incurred between the reAgent and server. This can be expressed as

$$D_{crs} = D_{cr} + pD_{rs} + \lambda$$

where p is the probability of a cache miss.

The speedup of a reAgent Cacher over client/server is

$$speedup = 1 - \frac{D_{cr} + pD_{rs} + \lambda}{D_{cs}}$$

and $speedup > 0$ when

$$(1 - p)D_{rs} > \lambda$$

So caching is better when

- D_{rs} increases (the delay from the network segment skipped by the cache is higher)
- p decreases (fewer cache misses)
- λ decreases (overhead of using reAgent decreases)

Thus, a reAgent cache outperforms client/server when the delay between reAgent and server is high enough, or the miss rate is low enough to overcome reAgent overhead.

V.D Collator

Description The Collator behavior (Fig. V.4) transmits the same message to multiple servers from a remote location, and waits until a wait condition, specified by the client, is satisfied. Afterwards, the responses are sent to an application-specific function that produces a result for the client (collating).

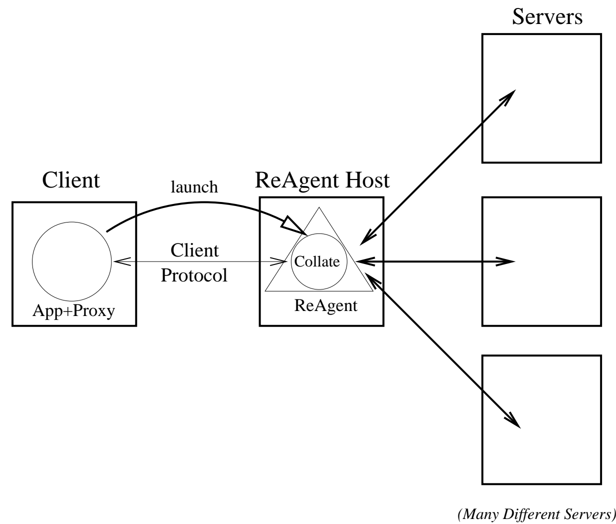

Figure V.4: The Collator Behavior

Base Logic

```
input: request, serverList

----------------------------------------
n = sizeof(serverList)

replies = 0    // synchronized

for (i = 1 to n)
  spawn Thread that runs :
  for (s = 1 to n)
    serverProtocol.connect(serverList[i])
    serverProtocol.send ()
    response[i] = serverProtocol.receive()     // blocking until timeout
    replies = replies + 1

collateLogic.wait ();

fusedResponse = collateLogic.collate(response) // all responses passed
----------------------------------------

output: fusedResponse
```

The message is sent once to the reAgent, which then transmits it multiple times, once for each server. The reAgent then waits for responses from the servers in an application-specific fashion, as defined by the `wait()` method (the CL). For example, the reAgent may only wait for the first response from any server, or for some bounded number of responses, or even wait for responses from the servers within a timeout period. After the `wait()` method returns, the server responses

are collated by the `collate()` function (also part of the CL), and the result is sent to the client.

Application A typical Web application that exhibits this behavior is a comparison agent that queries different servers with the same question and returns the "best" result. While many services for finding the best price of an item already exist on the Web, they do not perform correctly if a server is not known or supported by the query service, or if the user is more concerned about some other attribute, such as delivery time or seller reputation, that the service does not support.

Analysis For ease of analysis, the following assumes that the messages are sent serially, not in parallel.

Under straight client/server, the delay for completing requests to n servers is

$$D_{cs} = n \times d_{cs}$$

where d_{cs} = the average delay between client and server.

Under a collating reAgent, the time is

$$D_{crs} = nd_{rs} + \lambda$$

where d_{rs} = the average delay between reAgent and server.

Applying the speedup equation (V.1) gives a speedup of

$$speedup = 1 - \frac{nd_{rs} + \lambda}{nd_{cs}}$$

and a performance win for reAgents when

$$(n-1)d_{cr} > \lambda$$

where d_{cr} = the average delay between client and reAgent.

Thus, for any advantage to be gained, $n > 1$ (confirming the obvious). For each n beyond 1, a collating reAgent reduces delay by d_{cr}, which logically corresponds to traversing the network segment between the client and the reAgent, per additional server. If this amount is greater than the overhead from using the reAgent, the reAgent performs better than client/server.

V.E Summary

In this chapter we described effective general behaviors. Then, for each general behavior, we verified and quantified its advantages. Now it remains to show how one can use these behaviors as templates to develop client-specific reAgents. In the next chapter, we begin our description of the external programming interface used to implement these behaviors.

This chapter, in part, has been submitted for publication in the 5th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS06) under the title "ReAgents : Behavior-based Remote Agents and Their Performance". The dissertation author was the primary researcher and author of this paper, and the co-author Joseph Pasquale directed and supervised the research which forms the basis for this chapter.

Chapter VI

ReAgent API

In this chapter, we describe the API accessible to a reAgent programmer working from the client side. Later chapters describe the internal interfaces that are of primary interest to systems programmers.

For each component that is visible to the application programmer, we define an interface. The visible components are Behaviors, Converters, Protocols, and the ReAgent component, which acts as a container for these components. While all of these components are language-independent, we have shown their implementation in the Java language, as our first implementation of reAgents was in Java. (Our reasons for this approach will be described in the implementation Chapter (Chapter IX)).

VI.A The `ReAgent` Interface

All reAgents implement the `ReAgent` container interface, which defines the following methods:

```
public interface ReAgent {
  public ReAgent (Protocol cp);

  public void addBehavior (Behavior b);
  public void addBehavior (Behavior b, Converter c);
```

```
    public boolean launch (String hostName, String configFile);
    public Object process (Object request);
    public void stop ();

}
```

Constructor method:

- `ReAgent(Protocol cp)` creates an environment which keeps track of all the behaviors supported by that reAgent and any common variables, such as the order of behaviors and the client communications handler. It takes an instantiation of the client protocol `cp` as an argument.

General methods:

- `addBehavior(Behavior b)` adds a general behavior `b` to the reAgent environment.

- `addBehavior(Behavior b, Converter c)` adds a general behavior `b` to the reAgent environment with a specific converter `c` associated with it.

- `launch(String hostName, String configFile)` moves the current reAgent to the environment of the machine designated by `hostName` (the reAgent host) using the parameters defined in the file `configFile`. It is provided in the general `ReAgent` API because all reAgents are assumed to use the same launch procedure (i.e. determine agent system, get reAgent host, set up reAgent, deploy).

- `process(Object request)` sends a `request` to the reAgent using the client protocol `cp`. Upon receipt by the reAgent, the request is processed by the behaviors in its environment, and the output of these behaviors is returned.

- `stop()` method takes no arguments and terminates the reAgent. This is provided in the `ReAgent` API because all reAgents are assumed to use the same termination procedure.

VI.B The Behavior Interface

In our design, each general behavior is a specific instance of the parent `Behavior` component. This component's interface is

```
public interface Behavior {
  public Behavior (String type, ServerProtocol sProtocol,
                   String CL, String[] CLargs);

}
```

The only method the `Behavior` supports is its constructor, which takes the type of behavior, the server protocol, the CL, and the CL's arguments as its parameters. The type of behavior defines the base logic used, and the customized methods supported. In addition to customized methods, a behavior's interface may also include *control methods*: methods that allow the client to override with the operation of the behavior during run-time.

VI.B.1 Customizing Logic Interfaces

The client is responsible for providing customizing logic that can communicate with the base logic of the general behavior. For example, a customized monitor must know how to interface with a `MonitorLogic` object, which is a child of the general `Behavior` class. Each behavior defines a corresponding Logic interface (i.e., `MonitorLogic`, `CacherLogic`) which the customizing logic must support.

GenericResponse Object Some of the logical interfaces use an abstract data structure that represents a server response: `GenericResponse`:

```
public class GenericResponse implements java.io.Serializable
{
    public Object response;
    public byte[] content;
}
```

The `GenericResponse` data structure splits out the content from the raw response data so that the general behavior can handle the response in an abstract fashion.

The FilterLogic Interface Customizing logic that acts as a Filter must support the following methods:

```
public interface FilterLogic extends Behavior {

  // called by the reAgent logic
  public byte[] filter (byte[] content);

  // may be called by the user for control
  public void setLevel(int level);
}
```

Customizing method:

- `filter(byte[] content)` reduces the server data (stored in `content`) in an application-specific fashion.

Control method:

- `setLevel(int level)` allows the client to dynamically adjust the degree of filtering according to the amount specified by `level`.

The MonitorLogic Interface Customizing logic that acts as a Monitor must support the following methods:

```
public interface MonitorLogic extends Behavior {
  // called by the reAgent logic
  public long calcNextQuery (Response responseStruct, long lastQuery);
  public boolean testResponse (String [] args, Response responseStruct);

  // may be called by the user for control
  public void sendQueryNow();
}
```

Customizing methods:

- `calcNextQuery(GenericResponse responseStruct, long lastQuery)` returns the next time the monitoring reAgent should make another query.
- `testResponse(String args, GenericResponse responseStruct)` tests to see if the server response (stored in `responseStruct` has produced a trigger state.

Control method:

- `sendQueryNow()` forces the monitoring reAgent to query the server immediately.

The CacherLogic Interface Customizing logic that acts as a Cacher must support the following methods:

```
public interface CacherLogic extends Behavior {
  // called by the reAgent logic
  public String hash (byte[] request);
  public boolean lookup (String key);
  public Response get (String key);
  public void replace (String key, Response responseStruct)

  // may be called by the user for control
  public void flush();
  public void changeCacheSize(int size)
}
```

Customizing methods:

- `hash(byte[] request)` takes a request as input and returns a String that is the key string for that request.
- `lookup(String key)` returns true if the key string `key` is in the cache.

- `get(String key)` returns the `GenericResponse` associated with key string `key` in the cache.
- `replace(String key, GenericResponse responseStruct)` puts key string `key` in the cache and associates it with a `GenericResponse`.

Control methods:

- `flush()` immediately empties the cache.
- `changeCacheSize(int size)` allows the client to dynamically modify the size of the cache to the amount specified by the parameter `size`.

The CollatorLogic Interface Customizing logic that acts as a Collator must support the following methods:

```
public interface CollatorLogic extends Behavior {
  // called by the reAgent
  public void wait ();
  public Object collate (Response[] responses);

  // called by the user for control
  public void forceCollate ();
}
```

Customizing methods:

- `wait()` pauses the reAgent until a certain condition (defined in the method) has been met.
- `collate(Response[] responses)` takes all the results received and combines them into one object to be sent back to the client.

Control method:

- `forceCollate()` forces the reAgent to exit the `wait()` method and begin collation immediately.

VI.C The Converter Interface

The `Converter` is the simplest reAgent component. It only needs to support one method:

```
public interface Converter {
  // convert function
  byte[] convert (byte [] response)
}
```

Method:

- `convert(byte[] response)` takes the output of a behavior (usually a server response) and converts it into a new server request for another behavior.

VI.D The Protocol Interface

The `Protocol` component is used for handling communications between reAgent and the outside world.

```
public interface Protocol
{
  public boolean connect (InetAddress address, int port);
  public Protocol waitForConnect (int port);
  public void send (Object obj);
  public Object receive ();
  public void disconnect ();
  public void cleanup();
}
```

The client and server protocols must be ported to this interface. (The specific implementation of a protocol, like customizing logic, is not part of the reAgent API, but common protocol implementations can be provided by a third-party developer.) Consequently, the interface only defines general functions that all

protocols must support (send, receive, connect, and disconnect). In this manner, flexibility of protocol choice is retained while giving the reAgent an interface that it can use to communicate with client and server.

The ServerProtocol Interface The `ServerProtocol` interface is a sub-class of Protocol that allows the code implementing the general behavior to interface with the specific server protocol. Consequently, all server protocols must be of type `ServerProtocol`. The interface, in code, is :

```
public interface ServerProtocol extends Protocol
{
  public Object handleRequest (byte[] request);
  public byte[] assembleResponse (byte[] content, Object oldResponse);
}
```

Methods:

- `handleRequest(byte [] request)`, turns a general request by client into the actual request to server. The server is then contacted with this request and its response, if any, is returned.
- `assembleResponse(byte [] content, Object oldResponse)`, allows the behavior to generate a new response in the format of the server protocol that functions as the behavior's output. The two arguments are the new data and the old response. This is a necessary abstraction when the reAgent changes the server response, so that the reAgent can assemble a new response without knowing the details of the server protocol.

VI.E Summary

This chapter has outlined the general methods that are either called by the application code or need to be implemented by the application programmer.

The next chapter will explain how these methods are used to create and deploy reAgents.

This chapter, in part, is a reprint of material as it appears in the Thirteenth International World Wide Web Conference (WWW2004) under the title "Web Customization Using Behavior-Based Remote Executing Agents". The dissertation author was the primary researcher and author of this paper, and the co-author Joseph Pasquale directed and supervised the research which forms the basis for this chapter.

Chapter VII

Programming Examples

To provide some intuition as to how reAgents work and simplify programming, we present some examples in this chapter, showing how reAgents are used and easily changed to meet to the user's requirements.

VII.A Examples of Usage

The implementation of reAgents is encapsulated in a package of files, known as the ReAgent package. The ReAgent package facilitates cooperation with existing mobile code systems by providing interfaces that will properly interact with that system to create an intermediary (the reAgent) that customizes the client/server communications according to the customizing logic. In order to create the reAgent, the developer must:

1. Choose the mobile code system (specified as a parameter in a configuration file), and a machine with that system installed, the *reAgent host*
2. Choose the reAgent behavior that describes the general behavior of the CL
3. Provide an implementation of the CL to run on the reAgent host
4. Provide an implementation of the server and client protocols

In all of the following examples, to simplify the discussion, we assume that the mobile code system has been pre-determined, and that the language of implementation is in Java.

VII.A.1 Example 1: Basic Data Filtering reAgent

Consider the problem of Web browsing on a PDA over a low-bandwidth connection to the Internet. Given that images can take a long time to download because of the limited bandwidth, and that the screen display is not big enough to accommodate large images, it would be beneficial to interpose a reAgent to shrink the image to a size that the PDA screen can support before it is sent over the low-bandwidth connection, catering to both usability and performance.

Using the reAgents Package

To support this common scenario, after choosing a mobile code system (step 1), the programmer picks the **Filter** behavior template from the behavior library (step 2). This is because shrinking shares the defining characteristic of filters: reducing the server data to a different format (for performance reasons) before transmission. The CL must also be implemented (step 3). For this particular example, the programmer bases the CL on a shrinking algorithm using standard Java image library functions. Here is the relevant class that implements this algorithm :

```
public static BufferedImage shrink(BufferedImage src, double factor) {
    int w = (int) (src.getWidth() * factor);
    int h = (int) (src.getHeight() * factor);
    Image image = src.getScaledInstance(w, h, Image.SCALE_AREA_AVERAGING);
    BufferedImage result =
        new BufferedImage(w, h, BufferedImage.TYPE_INT_RGB);
    result.createGraphics().drawImage(image, 0, 0, null);
    return result;
}
```

Using the FilterLogic Interface

The programmer must port the CL to the behavior-specific interface (in this case, the `FilterLogic` interface in our Java implementation) so that the Filter template knows how to invoke the CL.

As a reminder, the `FilterLogic` interface requires the CL to support the following methods:

```
public interface FilterLogic extends Behavior {

  // called by the reAgent logic
  public byte[] filter (byte[] content);

  // may be called by the user for control
  public void setLevel(int level);
}
```

The algorithm of the `shrink` function can be used as the `filter` function, with the `factor` argument being set by `setLevel`. Here is the `Shrink` class in full:

```
class Shrink implements FilterLogic {

    int shrinkFactor;

    // the private, internal custom logic
    private static BufferedImage shrink(BufferedImage src, double factor) {
        int w = (int) (src.getWidth() * factor);
        int h = (int) (src.getHeight() * factor);
        Image image = src.getScaledInstance(w, h, Image.SCALE_AREA_AVERAGING);
        BufferedImage result =
          new BufferedImage(w, h, BufferedImage.TYPE_INT_RGB);
        Graphics2D g = result.createGraphics();
        g.drawImage(image, 0, 0, null);
        return result;
    }
```

```
    // the public interface for using the private logic
    public byte[] filter(byte[] content) {
        BufferedImage sourceImage = new BufferedImage(content);
        targetImage = shrink(sourceImage, shrinkFactor);
        return targetImage.toBytes();
    }

    // the public interface for setting the shrinking facotr
    public void setLevel(int level) {
        shrinkFactor = level;
    }

}
```

Using the ReAgent API

Once the CL is ported to this interface, the client and server protocols need to be defined (step 4). For this example, we use the pre-defined standard protocols included with the ReAgents package: a simple plaintext protocol on top of TCP/IP[32] for the client protocol, and HTTP[16] for the server protocol. (More information on these protocols can be found in Section IX.C.) A filtering reAgent can then be created with the following code:

```
ReAgent reagent;
Protocol clientProtocol = new Plaintext();
ServerProtocol serverProtocol = new HTTP();

String filterCL   = "Shrink.class";
String[] filterArgs = {"320", "200"};
String configFile = "standard.cfg";

reagent = new ReAgent(clientProtocol);

Behavior filter = new Behavior("Filter", serverProtocol,
                               filterCL, filterArgs);
```

```
reagent.addBehavior(filter);

reagent.launch("tap.ucsd.edu", configFile);
```

The constructor method of `reAgent` creates a reAgent that will communicate with the client using the plaintext-based client protocol (included in the package). A filtering behavior is then added to the reAgent, with the appropriate CL and arguments. In this example, the CL takes as arguments the target image resolution of 320x200. Then, the code launches the reAgent to the location "`tap.ucsd.edu`", provided as a parameter. The reAgent is now at the remote host (the "reAgent host") and is ready to communicate with the client.

For the client to send the reAgent (and through it, the server) a request, the `process` method is invoked :

```
byte[] response = (byte []) reagent.process (request);
```

The `process` method is responsible for actually communicating with the reAgent. It takes a request to pass to the server as an argument and returns the output of the reAgent in a byte array. All the filtering is hidden from the user; the reAgent, through the pre-defined Filter classes, is responsible for calling the client-specific Filter function (to shrink the server data) when appropriate.

At this point, we have a working reAgent that compresses the server data before sending it to the browser. All the complexity of the actual movement, communications, and processing has been abstracted into a few lines of code. Neither the browser nor the server is aware of the filtering — the browser is connected to the reAgent, but is not privy to its internal function, while the server treats the reAgent as the actual browser. In this way, the reAgent homogenizes the client environment for servers while requiring no additional effort or knowledge on the server's part.

VII.A.2 Example 2: Customizing Communications

ReAgents, composed of CL-containing behaviors, abstract away movement and communications from the programmer. However, while movement is designed to be fixed to a one-shot style, similar restrictions are not placed on communications. Just as they are parameterized to accommodate specialized customizing logic functions, reAgents can accommodate special communications requirements by allowing specification of communication protocols that the reAgent uses to communicate with the client and server. Such custom protocols are useful whenever specialized communications are needed, such as in the case of an unreliable network whose default error recovery mechanism is unsatisfactory. (If custom protocols are not necessary, nothing special needs to be done, as default protocols are automatically provided when the constructor is called without a custom protocol as an argument.)

The `Protocol` interface (Section VI.D) defines only highly general functions that protocols must support (send, receive, connect, and disconnect). In this manner, flexibility of protocol choice is retained while giving the reAgent an interface that it can use to communicate with client and server in a customized fashion. Furthermore, by separating the single client/server protocol into client-reAgent and reAgent-server components, the reAgent is able to communicate with the client using a client-specific protocol that specifically addresses client problems unsupported by the server protocol.

In this example, the user of the PDA is in a hotel attending an academic conference. Balking at the exorbitant rates that the hotel is charging for high-speed Internet access, the user finds a nearby, unsecured wireless router and hooks up. However, this wireless network also happens to be extremely unreliable, dropping packets with great frequency. After implementing a customized error-recovery protocol (`StableProtocol`) to improve application performance over this network, linking the reAgent with the custom protocol uses the following code:

```
Protocol clientProtocol = new StableProtocol();
ServerProtocol serverProtocol = new HTTP();

String filterCL   = "Shrink.class";
String[] filterArgs = {"320", "200"};
String configFile = "standard.cfg";

ReAgent reagent;
reagent = new ReAgent(clientProtocol);

Behavior filter = new Behavior("Filter", serverProtocol,
                               filterCL, filterArgs);
reagent.addBehavior(filter);

reagent.launch("tap.ucsd.edu", configFile);
```

The reAgent created will then communicate with the client with the client protocol and with the server using the server protocol.

VII.A.3 Example 3: Integration into Traditional Applications

This section shows how a traditional browser application would integrate the previous two examples to use a filtering reAgent. First, here is the top-level code view of a traditional implementation of a HTTP Web browser:

```
Protocol HTTP = new HTTP();

public void main () {
  while (true) {
    request = getInputFromClient();        // gets input from keyboard
    HTTP.send(request);                    // server specified in request
    byte[] response = HTTP.receive();
    displayResponse (response);
  }
}
```

In order to transform the browser application into one using a reAgent that communicates with the client and server using HTTP, the following changes are made (changed lines marked with an asterisk) :

```
public void main () {

  Protocol clientProtocol = new HTTP();                     *
  ServerProtocol serverProtocol = new HTTP();
  ReAgent reagent;                                          *

  String filterCL   = "Shrink.class";                       *
  String[] filterArgs = {"320", "200"};                     *
  String configFile = "standard.cfg";                       *

  reagent = new ReAgent(clientProtocol);                    *
  Behavior filter = new Behavior("Filter", serverProtocol,  *
                               filterCL, filterArgs);      *
  reagent.addBehavior(filter);                              *
  reagent.launch ("tap.ucsd.edu", configFile);              *

  while (true) {
    request = getInputFromClient();
    byte[] response = (byte []) reagent.process (request);  *
    displayResponse (response);
  }
}
```

All communications and filtering are now completely hidden from the browser — the template abstracts them away with the `process()` method.

VII.A.4 Example 4: Changing the CL to Filter Ads

Little work was needed to create the filtering reAgent beyond obtaining a suitable CL and interfacing it with the appropriate behavior. However, one could

argue that one could simply build a filtering agent and use it for any client in a similar situation, such as shrinking images for small PDAs. But not all users need to shrink images.

Take another user, this time one who is unconcerned about image size, but who does not enjoy reading advertisements inserted into Web pages. Here, the user would find a filtering reAgent useful, but for a different reason. The reAgent is used to intercept the offending ads before they reach the client, strip them out of the data, and send the content without the ads to the client. Depending on the ad filtering algorithm, the level parameter can be used in a variety of ways, such as determining the maximum size of ads to allow without filtering, or counting the number of popups a site sends before the filter is activated.

Note that this is the same type of behavior as in the first example, except that instead of shrinking images, the reAgent is removing ads. In each case, a reAgent receives data from the server, changes it to conform to a client's custom requirements (using the level to determine the degree of change), and then sends it to the client. With reAgents, the shared parts of the behavior are already written. The programmer only needs to provide a suitable CL to customize the reAgent's behavior towards the needs of the client. For example, here the CL is chosen to be the popular AdBlock filter. An implementation of AdBlock is written in `AdBlock.class`, ported to the `FilterLogic` interface, and sent to the template as a parameter to create a reAgent that behaves in a predictable, useful fashion:

```
public void main () {

  Protocol clientProtocol = new HTTP();
  ServerProtocol serverProtocol = new HTTP();
  ReAgent reagent;

  String filterCL   = "AdBlock.class";                         *
  String[] filterArgs = {};                                    *
  String configFile = "standard.cfg";
```

```
    reagent = new ReAgent(clientProtocol);
    Behavior filter = new Behavior("Filter", serverProtocol,
                                   filterCL, filterArgs);
    reagent.addBehavior(filter);
    reagent.launch ("tap.ucsd.edu", configFile);

    while (true) {
      request = getInputFromClient();
      byte[] response = (byte []) reagent.process (request);
      displayResponse (response);
    }
  }
```

Note that the only changes from the previous filtering example are in the class file for the Filter and its arguments (the size of the ad).

VII.A.5 Example 5: Using Other Behaviors

In addition to reAgents providing flexibility in customizing logic with similar behaviors, reAgents also make it easy to use any customizing logic that conforms to one of the characteristic behaviors identified. The user simply needs to change the type of reAgent used to instantiate the agent and use a CL that conforms to the logic interface for that behavior.

In this example, a user has access to an online stock exchange, and wants to be alerted when the stock of MacroHard (ticker symbol: MHRD) drops to a target price of $100, but only after it has done so three times in a 24–hour period (known as a "triple dip" in technical analysis). The online brokerage doesn't offer this service, but the user can program a reAgent using a Monitor behavior to watch the stock price's movements. The user writes up his algorithm in a class called `TripleDip`. This class uses the `MonitorLogic` interface, and it takes as arguments the stock ticker symbol and the target price. Then, he writes the following code:

```
public void main () {

  Protocol clientProtocol = new StandardProtocol();
  ServerProtocol serverProtocol = new HTTP();
  ReAgent reagent;

  String monitorCL  = "TripleDip.class";                          *
  String[] monitorArgs = {"MHRD", 100};                           *

  String configFile = "standard.cfg";

  reagent = new ReAgent(clientProtocol);
  Behavior monitor = new Behavior("Monitor", serverProtocol,      *
                                  monitorCL, monitorArgs);        *

  reagent.addBehavior(monitor);                                   *
  reagent.launch ("tap.ucsd.edu", configFile);

  while (true) {
    request = getInputFromClient();
    byte[] response = (byte []) reagent.process (request);
    displayResponse (response);
  }
}
```

In this manner, the user is able to identify a triple dip without needing server support for this type of action. The changes from the filter example are straightforward, with only the behavior type and the custom logic changing.

VII.A.6 Example 6: Using Multiple Behaviors

ReAgents can be powerful when multiple behaviors are chained together. For example, take the monitoring reAgent of example 5. Now, instead of merely

being alerted, the user wants to automate the process of selling the stock when the stock experiences a "triple dip" (supposedly, a sell signal). Most online brokerages do not support such a specific request, but a reAgent can perform this task by linking the Monitor with a Filter that doesn't filter (one that merely relays a request and returns a response). The two behaviors are tied together with a `Converter` that transforms the server response back to an appropriate server request to sell the stock `GenerateSellRequest`. In this manner, when the Monitor finishes by detecting a "triple dip", the Filter is triggered to pass a sell request to the server and the stock is sold without paying the latency cost of transmitting the information to the user or the user reaction time. The code :

```
public void main () {

  Protocol clientProtocol = new HTTP();
  ServerProtocol serverProtocol = new HTTP();
  ReAgent reagent;

  String monitorCL  = "StockPoller.class";
  String[] monitorArgs = {"MHRD", 100};

  String convertCL  = "GenerateSellRequest.class";          *
  String filterCL   = "Relay.class";                        *
  String[] filterArgs = {};                                 *

  String configFile = "standard.cfg";

  reagent = new ReAgent(clientProtocol);
  Behavior monitor = new Behavior("Monitor", serverProtocol,
                                  monitorCL, monitorArgs);

  Converter fConvert = new Converter(convertCL);            *
  Behavior filter = new Behavior("Filter", serverProtocol,  *
                                  filterCL, filterArgs);    *
```

```
    reagent.addBehavior(monitor);
    reagent.addBehavior(filter, fConvert);                    *
    reagent.launch ("tap.ucsd.edu", configFile);

    while (true) {
      request = getInputFromClient();
      byte[] response = (byte []) reagent.process (request);
      displayResponse (response);
    }
  }
```

VII.B Best Practices

ReAgents are fairly straightforward to use correctly. In this section, we describe some best-practices approaches for using reAgents.

VII.B.1 Determining Location

Given an atypical network segment between client and server that is deficient in some manner beyond server expectations (the *critical link*), the reAgent host must be chosen so that when the reAgent communicates with the server, all such critical links are avoided or bypassed. This causes all communications between reAgent and server to be conducted using a typical network segment, which, by definition, are without atypical problems. It also allows the reAgent to communicate with the server according to server expectations, and with the client in a fashion that caters to the deficiencies of the critical link.

VII.B.2 Determining Protocols

To use a reAgent, two protocols need to be given as arguments to the reAgent, the server protocol and the client protocol. Determining the server protocol is simple: it should be an implementation of the protocol that the server uses,

ported to the `ServerProtocol` interface. For example, for most Web applications, the server protocol should be an HTTP `ServerProtocol` implementation.

On the other hand, the client protocol should be a protocol that caters in some fashion to the critical link. For example, if the critical link is a wireless connection with no security against eavesdroppers and the server protocol does not support a high level of encrpytion, a protocol which encrypts data with a stronger encryption mechanism could be used to eliminate the security hole.

VII.B.3 Determining Behaviors

Usually the choice of a behavior for a single-behavior reAgent is straightforward. The behaviors have been designed to be extremely different from each other in their patterns of communication and logic, and many common network applications are a specific implementation of a behavior (such as a Web filter being a Filter implementation).

However, sometimes a complicated operation is required by the reAgent developer, one that a single behavior can not satisfy. Behavioral determination for a multi-behavior reAgent is an issue that was not explored fully at the time of this dissertation, and is an area for future research.

VII.C Summary

This chapter presented several scenarios for using reAgents. We showed the ability of reAgents to facilitate changes in logic, communications, or behavior. We also showed how reAgents can link behaviors in sequence to create even more powerful reAgents. Finally, we explained some ideas to help a reAgent developer decide how to use reAgents effectively.

Chapter VIII

Internal API

ReAgents are designed to be interdependent of mobile code system. To provide a simple interface to the user by abstracting away differences in mobile code systems, reAgents must be configured to interface with an existing mobile code system before that system can be used. Alongside the client API, the Java reAgents package also defines an internal API, hidden from the client, that allows interaction with a Java-based mobile code system. The driving idea behind the design of this API has been the ability to support many agent systems with the reAgents code, because reAgents are supposed to be an abstraction that works across mobile code platforms.

VIII.A The AgentSystemHandler Interface

Because reAgents are conceptually independent of mobile code system, it is required to have an interface to the system, the `AgentSystemHandler`. The `AgentSystemHandler` interface allows the reAgent code to retrieve a handle to a mobile code system running on a remote machine, and to create and deploy a reAgent compatible with the system to that machine.

The `AgentSystemHandler` interface is defined as follows:

```
public interface AgentSystemHandler
{

  public abstract ReagentHost getReagentHost(String hostname)
    throws ServerNotFoundException;

  public abstract ReagentHandle deploy(Reagent reagent, ReagentHost iServer,
                                       String cProto, String sProto,
                                       URL[] codebase)
    throws RemoteException;

}
```

- `getReagentHost()` returns an handle to the host, of type `ReagentHost`.
- `deploy()` launches and starts the reAgent, and returns a handle to the reAgent that provides control and communication with any system protocol. The handle returned is of type `ReagentHandle`.

The two general interfaces `ReagentHandle` and `ReagentHost` allow the code to reference reAgents and their hosts in a general fashion. Thus, it does not matter if the mobile code is actually an IBM Aglet[25] or a Dartmouth D'Agent[17], the template code can treat it as a general reAgent via a `ReagentHandle`, and send it to a `ReagentHost` (instead of specifically an Aglet server or a D'Agents Server).

VIII.B The ReagentHandle Interface

`ReagentHandle` is an abstract class that should be the parent of any class that implements the handle that references a reAgent. The `ReagentHandle` class supports general methods for controlling and communicating with the reAgent:

```
public abstract class ReagentHandle {
  public abstract void start ();
  public abstract void stop ();
  public abstract void systemSend (Object request);
  public abstract Object systemReceive ();
}
```

- `start()` encapsulates the execution of a remote reAgent with this system.
- `stop()` encapsulates the termination of a remote reAgent with this system.
- `systemSend()` and `systemReceive()` encapsulate system-supported communication between the client and the reAgent.

With these methods, the template code is able to start, stop, and communicate with the reAgent – all functions that we consider fundamental to any reAgent. The communication capability, via what we call the *reAgent protocol* is a separate, internal communication mechanism from the client protocol specified by the client. The client protocol is used during normal operation and varies from reAgent to reAgent, but the reAgent protocol allows the client-side logic implementation to control high-level aspects of the reAgent in a standard fashion.

VIII.C The ReagentHost Interface

```
public abstract class ReagentHost {
}
```

`ReagentHost` is simply an empty interface that functions as the parent of any class that implements a reAgent host. The class does not define any methods,

it is merely an umbrella class for all reAgent host classes, which are implemented for a specific agent system, i.e., `AgletAgentHost` for Aglets.

VIII.D The ReAgent Protocol

This reAgent protocol consists of a few string-based commands that allows the `ReagentHandle` to control the reAgent, overriding its behavior(s). It is currently in a simple, primitive state, with only a few commands:

SEND_IP tells the reAgent to send back the IP address of its host.

ARG $< behavior\# >< args >$ changes the arguments of the behavior specified to *args*.

QUIT tells the reAgent to terminate prematurely.

On execution, the reAgent launches a thread to listen for commands sent by the client via the reAgent protocol. This enables the client to override the reAgent, or even change the behavior arguments during run-time.

VIII.E Summary

Because reAgents are theoretically extensible to all forms of mobile code, the implementation of the ReAgents package needs to be able to interact with reAgents and their hosts in a general fashion. The `AgentSystemHandler`, `ReagentHost`, and `ReagentHandle` interfaces are used by the ReAgents code to reference systems, hosts, and intermediaries in general, while the `Behavior` general logic allows us to port behaviors across mobile code platforms. We now examine the specific implementation of these interfaces within a mobile code system, the UCSD Java Active Extensions (JAE) system.

Chapter IX

ReAgent Implementation

Even with all these pre-defined interfaces, the reAgent design still leaves some room for implementation decisions. Among these areas are language, agent system, and protocols.

IX.A Language: Java

While the concepts behind reAgents are language-independent, the choice of language for an actual implementation affects deployability. We implemented reAgents in the Java language, because of its portability: its virtual environment, the Java Virtual Machine (JVM) provides a standard, homogeneous environment for execution that allows code to be compatible across platforms. Also, Java's excellent support for dynamic code loading and remote serialization of objects (for mobile code) [4] makes it the popular language of choice for mobile code systems, and allows us to leverage existing mobile code system technology by using these Java-based systems as a testbed for running reAgent-based applications.

IX.B Mobile Code System: JAE

In our implementation, we made use of a locally-developed mobile code system called Java Active Extensions (JAE) [29, 30]. We made this choice for

two major reasons: (1) it is a stripped-down, bare-bones Java implementation of one-shot code mobility, and (2) we were familiar with the system.

In order to integrate the reAgents package with the JAE system, specific implementations of `AgentSystemHandler`, `ReagentHost`, `ReagentHandle` are needed to generate code that works within the JAE system (called *extensions*). Thus, each reAgent must be implemented as an extension so that it can be deployed and executed on the remote extension server (the reAgent host). This chapter describes all of these implementations.

IX.B.1 The ExtensionHandler class

The `ExtensionHandler` class is a sub-class of `AgentSystemHandler`. It implements the methods `getReagentHost()` and `deploy()`.

In `getReagentHost()`, code for interacting with the Extension Manager is provided. The Manager provides handles to Extension Servers, and the handle it provides to the host is returned as an instance of `JAEHost` (a sub-class of `ReagentHost` (Section IX.B.3)).

In `deploy()`, the code loads a special extension written as part of the implementation, the `ReagentExtension`, that is the entry point to running any client-specific intermediary as an extension. The `ReagentExtension` is described in further detail in Section IX.B.4. The code then returns a `JAEHandle` (a sub-class of `ReagentHandle`) to the client for control and communication.

IX.B.2 The JAEHandle class

```
public class JAEHandle
  extends IntHandle
{
  ExtensionHandle ext;

  public JAEHandle (ExtensionHandle extension, String protocolName) {
```

```
  // instantiate the IntHandle class variables
  super(protocolName);

  // set the extension handle
  ext = extension;
}


/*
 * starts the extension
 *
 */
public void start ()
{
  try {
    ext.begin();
  } catch (RemoteException e) {
    e.printStackTrace();
  }
}


/*
 * stops the extension
 *
 */
public void stop ()
{
  this.systemSend("QUIT");
}


/*
 *
 * sending using the extensions communicator
```

```
   *
   */
  public void systemSend (Object request)
  {
    try {
      ext.send(request);
    } catch (RemoteException e) {
      System.err.println ("Error sending message via extension system.");
      e.printStackTrace();
    }
  }


  /*
   *
   * receiving using the extensions communicator
   *
   */
  public Object systemReceive ()
  {
    try {
      return ext.receive();
    } catch (RemoteException e) {
      System.err.println ("Error receiving message via extension system.");
      e.printStackTrace();
      return null;
    }
  }
}
```

The `JAEHandle` class defines an internal variable, `ext`, that allows the methods defined within the class to reference the extension as an `ExtensionHandle` object (part of the JAE API). Then, the `ExtensionHandle` interface is ported to the `ReagentHandle` interface by having its methods tied to `ReagentHandle`'s `start()`, `systemSend()`, and `systemReceive()` methods. In this manner, the

general reAgent code can handle extension objects without specific knowledge of extensions. (For more information on the `ExtensionHandle`, see [30]).

IX.B.3 The JAEHost class

```
public class JAEHost extends ReagentHost
{
  ExtensionServer eServer;

  public JAEHost (ExtensionServer server) {
      eServer = server;
  }
}
```

The `JAEHost` class is a data structure that defines an internal variable, `eServer`, that allows `deploy()` to reference the Server as an `ExtensionServer` object (also part of the JAE API). This variable is set in the implementation of `getReagentHost()` in `ExtensionHandler`. This allows the general reAgent code to operate without specific knowledge of the `ExtensionServer` object.

IX.B.4 The ReagentExtension class

```
public class ReagentExtension implements Extension, Serializable
{
  public ReagentExtension (Reagent reagent);
  public void run (Collection resources);

  public class Listener extends Thread
    public Listener(int port);
    public void run();

  public class BehaviorManager extends Thread
    public BehaviorManager (Protocol cProtocol);
    public void run();
}
```

The `ReagentExtension` class provides the hook for the extension server to run the client-specific code by implementing the `Extension` interface defined in the JAE API. In this interface, a `run()` method is called by the Extension Server when the extension is ordered to begin execution by a call to the `begin` method of the `ExtensionHandle` associated with the extension. The parameters to the extension (including the `ReAgent` object) are stored in memory, so that the reAgent can be transmitted from the client.

The `run` method of `ReagentExtension` is the main method of the extension, and it can communicate with the client via the reAgent protocol. This allows the client to control high-level aspects of the extension, such as changing the behavior parameters of the extension during run-time.

Listener sub-class

When the extension is ordered to listen, it starts a new thread using the `ListenerThread` class. This thread listens for network connections on a predefined port and, for each connection, spawns a new thread, the *behavior manager*, to communicate with it via the client protocol. The `ListenerThread` runs until ordered to shut down by the client via the reAgent protocol.

Behavior Manager sub-class

The behavior manager contains the logic that manages the behaviors that run on the reAgent host, and is activated when the client connects to the extension using the client protocol. The thread constructor takes the `ReAgent` passed into the `ReagentExtension` parameters and instantiates each behavior in its queue by linking the general behavior logic with its client-specific logic. The constructor also defines a source (a converter, which can be `null`) and sink (usually another converter) for each behavior. Each behavior receives a request from its source, which it processes by triggering a call to the server protocol at some point. Upon receipt and post-processing of the server response, the behavior outputs the re-

sponse to the sink, which is the source converter of the next behavior in the queue, linking the behaviors. An exception is the sink of the last behavior in the queue: since there is no behavior after this, the sink is not a converter, but the behavior manager itself. The behavior manager forwards this final response back to the client via the client protocol.

Once the inputs and outputs of the queued behaviors are linked, each behavior logic is started in a separate thread by the behavior manager. Requests from the client via the client protocol go through the behavior manager as input to the first behavior logic in the chain. The behavior manager then sleeps until a response from the last behavior logic in the queue is returned. Upon the return, the response is sent back to the client via the client protocol.

IX.C Standard Protocols

To enable the use of reAgents, it was helpful to develop two standard protocols that use the reAgent `Protocol` interface. The first implements a simple plaintext protocol, for basic debugging and testing of reAgent communications. The second implements HTTP, useful for writing reAgents in conjunction with Web applications.

IX.C.1 Plain-text Protocol Implementation

This is one of the simplest protocols that can be implemented using the `Protocol` interface, as all it does is take some data in the form of a byte array and write it to another instance of the class. This allows simple character strings to be used as entire messages, hence the name "plain-text protocol". There are no headers or codes that need to be parsed by the protocol handler; the content is the entire message.

The `Plaintext` class implements the `Protocol` methods in minimal fashion. `connect()` opens a socket to the IP address and port number parameters given

using the default Java libraries, while `waitForConnect()` allows incoming connections to use the `Plaintext` protocol. `send()` and `receive()` use byte arrays to store, send, and receive data, with no specialized handling beyond simple end-of-transmission detection. `disconnect()` and `cleanup()` merely close the opened sockets and any outstanding I/O streams.

A wrapper was also written to use the `Plaintext` protocol in communication with servers. The `ServerProtocol` methods are implemented as follows:

```
public Object handleRequest (InetAddress serverAddr,
                             int serverPort,
                             byte[] request) {

  if (connect(serverAddr, serverPort)) {
    send (request);

    byte [] responseBytes = (byte []) receive();

    GenericResponse response =
      new GenericResponse (responseBytes, responseBytes);

    disconnect();
    return response;
  } else {
    System.err.println("Server down, exiting.");
    return null;
  }

}

public byte[] assembleResponse (byte[] newData,
                                Object oldResponse) {

  return newData;
```

```
  }


}
```

The code is fairly straightforward. The `GenericResponse` object is constructed using the same data twice, because the content is exactly the same as the message itself. Even though in this instance the reAgent could simply parse the message as content, the reAgent has to use the `GenericResponse` methods in order to be able to interface with other protocols. Meanwhile, the `assembleResponse()` method simply returns the new data as the new response, since there are no headers to edit.

IX.C.2 HTTP Implementation

For this important protocol, we ported a simple Java implementation of HTTP [39] to use the `Protocol` interface. The HTTP package defines the `HttpInputStream` and `HttpOutputStream` classes, along with methods for parsing the data. Of particular interest in the port was the `send()` method, where certain headers (such as "If-Modified-Since", and "If-None-Match") needed to be removed to fool the server into sending a response:

```
public void send (Object request) {
  try {

    RequestBuffer newRequest = (RequestBuffer) request;

    newRequest.headers.removeHeader("If-Modified-Since");
    newRequest.headers.removeHeader("If-None-Match");
    newRequest.headers.removeHeader("If-Range");
    newRequest.headers.removeHeader("Range");

    // sends the actual request line
    newRequest.requestLine.print();
```

```
      out.writeRequest(newRequest, true);
      out.flush();
      debugMsg ("HTTP request sent.");
    } catch (Exception e) {
      System.err.println ("Error sending HTTP Request.");
      e.printStackTrace();
    }

    return;
  }
```

Meanwhile, the `receive()` method reads the HTTP response and splits it into content and headers. A `GenericResponse` object is created with these two as arguments, and is returned so that the reAgent can parse the content without knowing about the actual protocol:

```
  public Object receive () {

    ResponseBuffer httpResponse  = null;
    GenericResponse gResponse = null;

    try {
      httpResponse = in.readResponse();
    } catch (Exception e) {
      System.err.println ("Error receiving HTTP Response.");
      e.printStackTrace();
    }

    debugMsg ("Received HTTP Response.");

    gResponse = new GenericResponse(httpResponse.content, httpResponse);

    return gResponse;
  }
```

To have a reAgent use HTTP to communicate with the server, the two `ServerProtocol` methods had to be implemented to hide the internal workings of HTTP from the reAgent. The `handleRequest()` method parses the data for the server name and port and opens an HTTP connection:

```
public Object handleRequest (InetAddress serverAddr,
                             int serverPort,
                             byte[] request) {

  final int       DEFAULT_HTTP_PORT = 80;

  HttpInputStream httpInputStream = null;
  RequestBuffer   httpRequest     = null;
  GenericResponse response        = null;
  String          serverName      = null;

  //
  // turn the byte array of the request into an HTTP Request
  //

  httpInputStream = new HttpInputStream(new ByteArrayInputStream(request));

  try {
    httpRequest = httpInputStream.readRequest();
  } catch (Exception e) {
    System.err.println ("Error receiving HTTP Request.");
    e.printStackTrace();
  }



  //
  // parse out the server address and the server port
  //

  try {
```

```
  // there are two methods for a server to be specified in an HTTP request:
  //   1. is when the host is stated in a request header
  //      so check to see if hostname was given in request header
  serverName = httpRequest.headers.getValue("Host");

  if (serverName == null) {
    // 2. is when the hostname is given in the request
    serverName = httpRequest.requestLine.url.getHost();
  }

  serverAddr = InetAddress.getByName(serverName);
} catch (java.net.UnknownHostException uhe) {
  System.err.println("No IP address found for server " + serverName);
}

// if no port is specified, set to 80, the default HTTP port
if ((serverPort = httpRequest.requestLine.url.getPort()) == -1) {
  serverPort = DEFAULT_HTTP_PORT;
}


//
// transact with server --
// connect, send, receive, disconnect, return response
//

if (super.connect(serverAddr, serverPort)) {
  send (httpRequest);
  response = (GenericResponse) receive();
  disconnect();
  debugMsg ("Received HTTP response from server.");
  return response;
} else {
  System.err.println("Server down, exiting.");
```

```
      return null;
    }

  }
```

while the `assembleResponse()` method re-writes the Content-Length header to be consistent with any response that the reAgent provides:

```
  public byte[] assembleResponse (byte[] newData,
                                  Object oldResponse) {

    ResponseBuffer newResponse = null;
    byte [] buf = null;

    //
    // test if the oldResponse object is an HTTP response
    //
    if (! (oldResponse instanceof ResponseBuffer)) {

      System.err.println("oldResponse is not an HTTP response.");

    } else {

      newResponse = (ResponseBuffer) oldResponse;

      newResponse.content = newData;

      // adjust the content-length header to match the new content's length
      newResponse.headers.removeHeader("Content-Length");

      newResponse.headers.addHeader(
          new Header("Content-Length",
                     (new Integer(newData.length)).toString()));

    }
```

```
    //
    // convert the response to a byte array
    //

    ByteArrayOutputStream bas = new ByteArrayOutputStream();
    HttpOutputStream hos = new HttpOutputStream(bas);
    try {
      hos.writeResponse(newResponse);
      hos.flush();
    } catch (IOException e) {
      System.err.println ("Error converting response to byte array.");
    }

    return bas.toByteArray();

  }
```

Through the use of these interfaces, the reAgent logic is able to parse and send HTTP requests and responses in a general fashion.

IX.D Summary

To make the reAgent design a reality, the reAgents were implemented in Java to use a simple mobile code system, Java Active Extensions (JAE). by filling out a few interfaces with specific code to handle JAE structures. A similar implementation can be done to port reAgents to other agent systems such as Aglets. To enable basic reAgent communications, two protocols, a simple plain-text protocol and an HTTP protocol, were implemented using the `Protocol` and `ServerProtocol` interface.

Chapter X

Experiments

Having described a specific implementation of reAgents on the JAE system, we now describe several basic experiments conducted from implementing reAgents as Java Active Extensions. The experiments were chosen to show that they exhibit the goals of our design as described in Chapter IV; that is, that they are :

- Scalable — client-based approach allows for customization in communications without affecting server (Experiment 1)
- Effective — applications using reAgents are more effective when compared with traditional applications (Experiments 2 and 3)
- Deployable — not only are they easy to use and deploy, but their overhead is not significant (Experiment 3)

X.A Experiment 1: Scalable Communications

ReAgents provide scalable customization in both communications and operations. In both cases the client can customize the type of communication or operation without requiring knowledge on the server programmer's part.

To demonstrate the scalability of custom reAgent protocols, we implemented a bandwidth regulating algorithm on top of the Plaintext protocol, and

then constructed a reAgent that uses this protocol to regulate the bandwidth of the client/reAgent connection. We then tested the reAgent by using it with a simple client application that retrieves a large file from a data server.

Note that this is merely an example to demonstrate how a reAgent could used to implement a custom protocol. This is not to imply that traffic shaping by the server is undesirable or necessary. But if the client application needs traffic shaping that the server does not support, then the reAgent can add this functionality for the client.

The bandwidth regulator algorithm performs flow control on its input stream by limiting the bandwidth to not exceed a client-specified maximum number of kilobytes per second (passed to the module as a parameter). The regulator operates in cycles. At the beginning of a cycle, it checks the input stream for as much data as is allowed to be transferred in a cycle. It then sends what it has to its output stream, computes the time it took to send, and sleeps for the rest of the cycle. If the sending time overruns the cycle time, the next cycle begins immediately. Consequently it is desirable to have a cycle time larger than the expected sending time, but not too large so as to compromise performance. In our implementation, we have defined the cycle time as 100 ms.

By incorporating the regulating algorithm into the `send()` and `receive()` methods, we generated a protocol for a reAgent that would restrict bandwidth according to a designated level. This protocol could then be used in any client application that needs to regulate bandwidth in a client-specified manner.

Results

The user launched the reAgent to a chosen reAgent host for execution. The user, through the reAgent, then requested a 14668 kB image file from a server. The results are shown in Figure X.A.

The bandwidth used was consistently 20 kB/s, as requested by the client, except for the final segment of 8 kB, which was the proper remainder from dividing

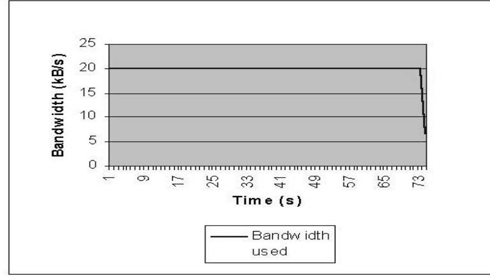

Figure X.1: Bandwidth Regulator Output

the file into 20 kB segments. To verify, this file was requested 24 more times, with identical results. Furthermore, the average transfer time was regularly within a millisecond or two of the expected transfer time, showing that the overhead from the regulator did not significantly impact performance.

Analysis

The experiment showed that the reAgent `Protocol` interface could be used to regulate the bandwidth for a low-bandwidth connection, without incurring much overhead. While not all aspects of deployability were covered in this experiment, the modularity of the `Protocol` interface allows only clients that need to regulate bandwidth the capability to do so. In addition, the server was completely unchanged during this experiment, maintaining compatibility with both regulating and non-regulating clients and increasing the scalability of a reAgent-based solution.

X.B Experiment 2: Effectiveness of reAgent Paradigm

In addition to providing greater scalability of customization, reAgent-based applications must gain a strong advantage over traditional applications, or there is no incentive behind their use. This section describes a couple of experiments to highlight the effectiveness of reAgents by using a stock trading application

(described in Section VII.A.5).

As a preliminary, a primitive stock server was developed as a testbed. The server outputs a ticker, showing the fluctuating price of a stock, and is programmed to accept only the first buy order it receives per run (i.e. only 1 share is available at this listed price).

The Monitor behavior was then used as the basis for a reAgent that would connect via network sockets to the stock server to watch stock prices and issue a buy order. The Monitor behavior was chosen because this application requires multiple communications between the reAgent and stock server. The reAgent monitors the ticker and sends a buy request when the price of a certain stock falls to a certain value (both chosen by the user as input arguments). The decision-making component is a custom algorithm that acts as the CL for the Monitor behavior.

In the following experiments, the performance metric used was the time it took for the server to receive a "buy" command after the stock hit the target price, essentially a measure of the total latency between the decision-maker and the server, plus the CPU time alloted to the decision-maker.

X.B.1 Experiment 2a: Response Time Comparison

The first experiment was to evaluate the relative performance of three network application paradigms :

1. traditional RPC – the Traditional paradigm
2. reAgent-based computing – the reAgent paradigm
3. customized incorporation of services – the Service paradigm

To introduce a non-trivial latency, the client machine, *federation*, a Sun Ultra 1, was stationed at Carnegie Mellon University in Pittsburgh, approximately 2500 miles away from the stock server machine, *ursus*, a Sun Ultra 10, in San

Diego. The results, after 10000 iterations for each paradigm, are summarized in the following table:

Table X.1: Comparison of Response Time for Various Paradigms

Paradigm	Response Time (ms)			Confidence (95%)	Standard Deviation
	Min	Max	Mean		
Traditional	71	3474	79.6	$\pm$ 1.8 ms	91.4 ms
ReAgent	1	14	1.161	$\pm$ 0.008 ms	0.39 ms
Service	0	11	0.051	$\pm$ 0.005 ms	0.27 ms

The confidence intervals show that the averages are statistically significant. These results say, not surprisingly, that the best performance occurs if the server can be hard-wired with the action that the client requests. While this might be true for simple operations like limit-order stock purchases, in practice, not all servers will be able to anticipate every need of the client, such as in the situation described in Section VII.A.5, where the user has an unusual stock-trading algorithm to implement.

Comparing the non-hardwired methods of traditional vs. reAgent, the traditional method suffers severe performance penalties as well as a highly variable response time. The high variance is a result of intermittent network congestion and timeouts, which the reAgent, running near the server, can ameliorate.

From this experiment, it can be concluded that a Monitor-based reAgent application achieves similar, although slightly inferior, performance when compared with a server-based service. However, the cost of building applications with a similar level of performance with reAgents is much smaller than having every server in the world hardwire their services for each user. Attempts to create flexible applications without reAgents result in highly variable and inferior performance due to the high-latency communications between the decision-making algorithm and the server.

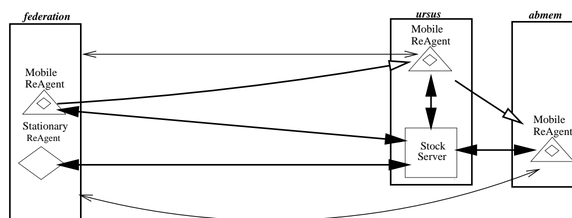

Figure X.2: Stock Traders Setup

X.B.2 Experiment 2b: Application Performance Comparison

An application-based comparison was performed as a supplement to the previous experiment. The comparison centered on the Monitor-based stock trading reAgent described in the previous section. This reAgent was run in sets of two. In each set, one reAgent remains at the client (the stationary reAgent), while the other reAgent (the mobile reAgent) is potentially launched to a different host to improve application performance. Each Trader's goal is to wait until the stock hits a pre-determined low price, and then buy it before their opponent.

The experiment was conducted with three sets of reAgents. The first set has both reAgents begin at *federation* (the client), to demonstrate the worst case of no supporting reAgent servers. The next set has the mobile reAgent moves to *ursus*, the stock server itself, and results are compared. The final set has the mobile reAgent move to *abmem*, a machine on *ursus*'s LAN that acts as a reAgent host. The last experiment is important because it highlights the deployable nature of reAgents; one can neither assume that every server in the world will support reAgents, nor that servers will always have the capacity to host reAgents. However, it is reasonable to assume that an independent reAgent host will be nearby.

The experiment showed that the mobile reAgent beat the stationary reAgent every time when operating from a closer location. The full results can be seen in Table X.2.

The 100% success rate in the cases where the reAgent migrated closer demonstrated that the mobile reAgent could parlay the theoretical performance

Table X.2: Mobile ReAgent vs. Stationary ReAgent

Mobile ReAgent Host	Mobile ReAgent Success Rate	95% Confidence Interval
Client	50.8%	$\pm$ 1.4%
Server	100.0%	$\pm$ 0.0%
ReAgent host	100.0%	$\pm$ 0.0%

benefits of reAgents into a superior, flexible application. In the worst-case scenario with no supporting reAgent servers, the reAgent still functioned on a relatively equal footing with a normal, traditional trader (represented by the stationary reAgent). However, the mobile reAgent was able to use its ability to move when closer reAgent hosts were available, allowing it to improve its performance along with its location. The reAgent allows this improvement to be easily customizable, merely by changing its destination (a line of text). As users have different requirements — for example, running directly on the server may be significantly more expensive, so a grad student's trading application may only give a destination of a cheaper reAgent server — reAgents allow users to scale the performance effectiveness to a level appropriate with their needs and abilities.

X.C Experiment 3: Deployability Overhead

The final design goal of reAgents is that they are highly deployable. While we were unable to conduct usability surveys, we were able to measure the overhead incurred by deploying reAgents. We implemented a simple filtering application, based on the Filter behavior, and show that the overhead is low, especially when taken relative to the performance gains derived by a filtering reAgent.

In this experiment, the filter simply reduced in size the Web server images received before sending it over a low-bandwidth connection to the client. The algorithm used was part of the ACME Labs JPM package [1].

Environment

In the following experiment, the following conditions applied:

- The client was a home computer PC PII–300 connected to the Internet via a dialup connection.
- The reAgent host was `tap.ucsd.edu`, a machine with 2 800Mhz Pentium III processors and on the same subnet as the data server.
- The data server was `charlotte.ucsd.edu`, the departmental web server.

The client was connected to the reAgent host via a dialup connection with effective bandwidth measured at 10–15 KB/s (KB = kilobytes). The reAgent host and data server were on the same subnet, so there was little overhead from network latency (thus allowing us to isolate observed overhead to our system). The regular bandwidth between the reAgent host and the server was measured at approximately 800 KB/s.

A fixed cost that needs to be paid at least once per reAgent creation is the launch overhead (the time it takes for the reAgent to be launched from the client to the reAgent host). The mean launch overhead of the JAE system for sending the reAgent and its associated classes over the local subnet was 984 ms (with a 95% confidence interval of 11 ms). Note that this is a one-time start-up cost; once the reAgent is launched, it can be used for multiple transactions, each of which involves receiving a request from the client, passing it to the server, getting the server's response, applying a function (in this case, filtering), and sending it to the client.

Setup

To eliminate alternative sources of overhead, a primitive Web browser was written in Java. It takes a series of HTTP requests as input, and returns the HTML output. The HTTP requests were for actual image files on the Web,

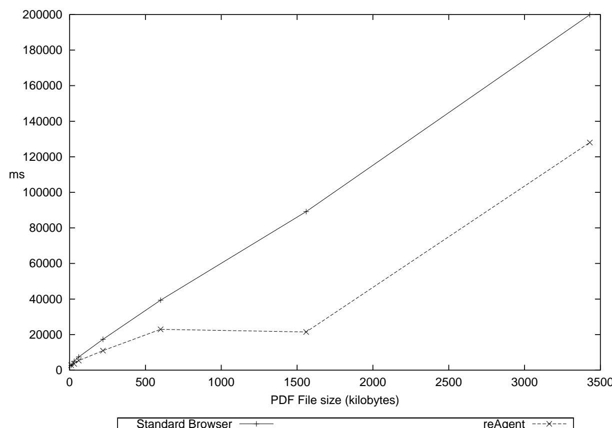

Figure X.3: End-to-end comparison times

ranging from 10 KB to 3.4 MB in size (with each successive file larger than the previous by a factor of approximately 2–3). This was to give the test suite a variety of realistic data files, which exhibited different filtering ratios, rather than canned ones that might be biased in favor of certain client-specific algorithms.

A proxy was also written as the interface for the browser to communicate with the reAgent. The browser was set to use the proxy, and launched the reAgent via a form. Thereafter, all communications from the browser went through the proxy and reAgent before the server.

Results

The results, compared to a non-filtering Web browser, are summarized in Fig. X.3. For most of the files, the filtering reAgent exhibited good performance gains, reducing end-to-end times by 30–75%. (The variable performance gain was dependent on how effective the reduction was.) The exception was the 10KB file, where the gain from compressing the data sent over the limited bandwidth link did not compensate for the reAgent processing overhead. However, the filter provided superior performance to the client/server approach for files greater than 10KB. An obvious optimization would be for the reAgent to not filter small files, as the benefit does not outweigh the cost.

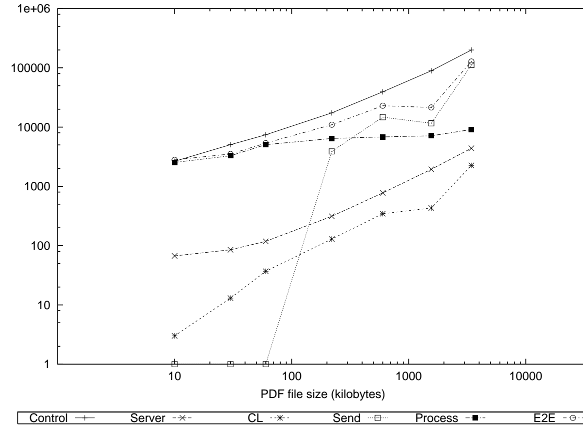

Figure X.4: Overhead of filtering (log-based)

We timed different parts of the reAgent while filtering in order to determine which factors are contributing the most to the end-to-end processing time and how they scale. The results are shown in Fig. X.4, which provides more detail for the smaller contributers to overhead by using logarithmic scales. The majority of the time was spent sending the data over the low-bandwidth link. The cost of filtering, processing, or moving the data from the server to the reAgent host were all minor and scaled well.

In this figure,

- **Control** is the end-to-end processing time for a standard (i.e., non-reAgent) client/server implementation.
- **Server** is the time it took for the reAgent host to download the file from the server (over a typical connection with high bandwidth).
- **CL** is the time for the filtering CL to operate.
- **Send** is the time it took for the client to download the file from the reAgent host.
- **Process** is the processing time of the reAgent on the file.
- **E2E** is the end-to-end processing time of the reAgent.

This experiment shows that the main bottleneck is clearly the network, and not the reAgent. In such cases, a reAgent based on the Filter behavior not only imposes little overhead (and hence high deployability), but can provide significant improvements in performance over a traditional client/server application.

X.D Summary

With these three experiments, we have shown that reAgents have met some aspects of the three design goals of scalability, effectiveness, and deployability. While none of the experimental results are surprising, they support and validate the analytical approach behind the reAgents model.

This chapter, in part, has been submitted for publication in the 5th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS06) under the title "ReAgents : Behavior-based Remote Agents and Their Performance". This chapter, in part, is also a reprint of material as it appears in the Thirteenth International World Wide Web Conference (WWW2004) under the title "Web Customization Using Behavior-Based Remote Executing Agents". The dissertation author was the primary researcher and author of these papers, and the co-author Joseph Pasquale directed and supervised the research which forms the basis for this chapter.

Chapter XI

Conclusion

In this dissertation, we described the reAgent (remote agent) programming abstraction for designing improved-performance client/server-based Internet applications. ReAgents are especially applicable to situations where clients are resource-limited in their computing or network access capabilities. A reAgent effectively extends the client's reach into the network, and derives its power by performing remote operations that :

1. minimize the effects of a problematic network path,
2. gain an advantage by operating close to one or more servers,
3. take advantage of remote resources (e.g., computational, memory) that are relatively more powerful than those locally available at the client,
4. act autonomously on behalf of the client in a customized fashion.

The use of reAgents allows us to derive benefits of mobile agent systems — specifically their support for dynamically-located remote execution, limited to one-shot movement — in an easy-to-program form. Key to this simplicity is the transparent handling of data migration and run-time network communications. This is a direct result of the reAgent programming model: reAgents are composed of one or more behaviors, i.e., abstractions for common remote patterns of action,

each of which can be specialized with custom logic that allows reAgents to be tailored to their applications. We identified four such general behaviors — filtering, caching, monitoring, and collating — and presented analytical models showing the conditions under which performance will improve when compared to the straight client/server model.

Our experience to date with reAgents has been in the context of Web-based applications [21], and mobile-computing applications [22]. We have found that they indeed lead to higher performance when conditions as predicted by the analytic models, most common in wireless environments with resource-limited clients, hold.

The main contributions from this work are as follows:

- Restricting movement of reAgents to one hop does not significantly impact the ability to construct useful, desirable applications. Meanwhile, it greatly simplifies security concerns and operation semantics.
- ReAgents can be categorized as behaving in a certain manner. We have identified a small core set of behaviors that capture common and useful patterns of action by remotely executing agents. These behaviors include the following: Filter, Monitor, Cacher, and Collator.
- We can more easily build agent-based applications through behavior templates. Behavior templates allow the programmer to plug in application-specific customizing logic to create a reAgent that customizes performance in a manner that fits their needs. This is a simple, scalable, and practical solution to the problem of client heterogeneity that adds little overhead.

Future avenues of research include identifying and implementing more behaviors, obtaining performance numbers for other basic template implementations, dynamic server protocol allocation, more complicated pre-defined protocols, and exploring the possibility of dynamically combining behavior templates to easily create applications with more complicated behaviors.

Bibliography

[1] Acme Labs, *JPM package*, `http://www.acme.com/java`, 1996.

[2] D. S. Alexander, W. A. Arbaugh, et al. *The SwitchWare Active Network Architecture*, IEEE Network, May/June 1998.

[3] Y. Aridor and D. B. Lange, *Agent Design Patterns: Elements of Agent Application Design*, Proc. of the 2nd Int'l Conference on Autonomous Agents, May 1998.

[4] K. Arnold and J. Gosling, *The Java programming language*, Addison-Wesley, Reading, MA, 2nd ed., 1998.

[5] R. Barrett and P. P. Maglio, *Intermediaries: New places for producing and manipulating web content*, Proc. of the 7th Int'l World Wide Web Conference, Brisbane, Australia, 1998.

[6] G. Banavar, T. D. Chandra, B. Mukherjee, J. Nagarajarao, R. E. Strom, and D. C. Sturman, *An efficient multicast protocol for content-based publish-subscribe systems*, Proc. of the 19th IEEE International Conference on Distributed Computing Systems (ICDCS '99), 1999.

[7] R. T. Boyer, and W. G. Griswold, *Fulcrum — An Open-Implementation Approach to Internet-Scale Context-Aware Publish / Subscribe*, Proc. of the 38th Annual Hawaii International Conference on System Sciences (HICSS '05), January 2005.

[8] P. Cao, J. Zhang, and K. Beach, *Active Cache: Caching Dynamic Contents (Objects) on the Web*, Middleware '98, September 1998.

[9] A. Carzaniga, D. S. Rosenblum, and A. L. Wolf, *Achieving scalability and expressiveness in an internet-scale event notification service.* 19th Proc. of the 19th ACM Symposium on Principles of Distributed Computing (PODC2000), 2000.

[10] V. Cerf, et. al, *Delay-Tolerant Network Architecture: The Evolving Interplanetary Internet.* IRB-TR-02-010, July 2002.

[11] G. Chen, and D. Kotz, *Solar: Toward a Flexible and Scalable Data-Fusion Infrastructure for Ubiquitous Computing.* UbiComp 2001 UbiTools Workshop, 2001.

[12] D. Chess, B. Grosof, C. Harrison, D. Levine, C. Parris, G. Tsudik, *Itinerant Agents for Mobile Computing* IEEE Personal Communications, 2(5): 34–39, 1995.

[13] P. Deutsch and J-L. Gailly, ZLIB Compressed Data Format Specification version 3, RFC 1950, Aladdin Enterprises, May 1996.

[14] M. Fry, and A. Ghosh, *Application Level Active Networking*, Computer Networks, 31(7): 655–667, 1999.

[15] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, October 1994.

[16] J. Gettys, J. Mogul, et al., *Hypertext transfer protocol*, HTTP/1.1, 1999 `http://www.w3.org/Protocols/rfc2616/rfc2616.html`

[17] R. Gray, G. Cybenko, et al., *D'Agents: Applications and Performance of a Mobile-Agent System*, Software - Practice and Experience, 32(6):543–573, May 2002.

[18] R. Gray, G. Cybenko, D. Kotz, and D. Rus, *Mobile agents: Motivations and State of the Art*, Handbook of Agent Technology, AAAI/MIT Press, 2002.

[19] R. Gray, D. Kotz, et al., *Mobile agents for mobile computing*, Proc. of the 2nd Aizu Int'l Symp. Parallel Algorithms/Architectures Synthesis, Fukushima, Japan, March 1997.

[20] C. Harrison, D. Chess, A. Kershenbaum, *Mobile Agents: Are They a Good Idea?*, IBM Research Report, March 1995.

[21] E. Hung and J. Pasquale, *Web Customization Using Behavior-Based Remote Executing Agents*, Proc. of the 13th Int'l ACM World Wide Web Conference (WWW2004), May 2004.

[22] E. Hung and J. Pasquale, *Using Behavior Templates To Design Remotely Executing Agents for Wireless Clients*, Proc. of the 4th IEEE Workshop on Applications and Services in Wireless Networks, August 2004.

[23] E. Hung and J. Pasquale, *ReAgents: Behavior-based Remote Agents and Their Performance*, Proc. of the 5th Int'l Joint Conf. on Autonomous Agents and Multi-Agent Systems, Hakodate, Japan, May 2006, to appear.

[24] D. Kotz, R. Gray, and D. Rus, *Future Directions for Mobile-Agent Research*, IEEE Distributed Systems Online, 3(8), August 2002.

[25] D. Lange, M. Oshima, G. Karjoth, and K. Kosaka, *Aglets: programming mobile agents in Java*, Proc. of Worldwide Computing and its Applications (WWCA'97), Lecture Notes in Computer Science, Vol. 1274, 1997.

[26] R. Lahoti. *Continuations in Java*, Master's Thesis, University of California, San Diego, 2003.

[27] A. Luotonen and K. Altis, *World-Wide Web proxies*, Computer Networks and ISDN Systems, 27(2): 147–154, 1994.

[28] P. Mohapatra, H. Chen, *WebGraph: A Framework for Managing and Improving Performance of Dynamic Web Content*, IEEE Journal On Selected Areas in Communications, 20(7), September 2002.

[29] T. Newhouse, *Java Active Extensions: A mobile-code mechanism for extending client resources*, Master's Thesis, UCSD, 2001.

[30] T. Newhouse and J. Pasquale, *Java Active Extensions: Scalable Middleware for Performance-Isolated Remote Execution to Enhance Wireless Network Applications*, Elsevier Computer Communications Journal, Volume 28, Issue 14, pp. 1680–1691, September 2005.

[31] Open Mobile Alliance, *Working Group (Affiliate) Specifications*, `http://www.openmobilealliance.org/tech/affiliates/index.html`, 2002.

[32] J. Postel, *Transmission Control Protocol (TCP)*, Internet RFC 793, `http://www.faqs.org/rfcs/rfc793.html`, 1981.

[33] R. Rivest, A. Shamir, and L. M. Adelman: *Cryptographic Communications System and Method*, US Patent 4,405,829, 1983.

[34] J. H. Saltzer, D. P. Reed, and D. D. Clark, *End-to-end arguments in system design*, ACM Trans. Computer Systems 2 (4): 277–288, November 1984.

[35] A. Silva, J. Delgado, *The Agent Pattern for Mobile Agent Systems*, European Conference on Pattern Languages of Programming and Computing, EuroPLoP, 1998.

[36] S. da Silva, D. Florissi, and Y. Yemini. *Composing active services in NetScript*, DARPA Active Networks Workshop, March 1998.

[37] A. Singh, R. Sankar, V. Jamval, *Design Patterns for Mobile Agent Applications*, Workshop on Ubiquitous Agents on Embedded, Wearable, and Mobile Devices, Bologna, Italy, 2002.

[38] J. W. Stamos and D. K. Gifford, *Remote Evaluation.* ACM Transactions on Programming Languages and Systems, 12(4):537–565, March 1990.

[39] J. Steinberg, *Java HTTP Protocol Implementation*, 2002.

[40] J. Steinberg and J. Pasquale, *A Web Middleware Architecture for Dynamic Customization of Content for Wireless Clients*, Proc. of the 11th Int'l World Wide Web Conference, Honolulu, Hawaii, USA, May 2002.

[41] D. Tennenhouse and D. Wetherall, *Towards an active network architecture*, Computer Communications Review, 26(2): 5–18, April 1996.

[42] A. Vahdat, M. Dahlin, T. Anderson, A. Agarwal, *Active Names: Flexible Location and Transport of Wide-Area Resources*, Proc. of the USENIX Symposium on Internet Technologies and Systems (USITS), October 1999.

[43] Y. Villate, A. Illaramendi, E. Pitoura, *Mobile and External Storage Space Using Agents for Users of Mobile Devices*, Workshop on Ubiquitous Agents on Embedded, Wearable, and Mobile Devices, Bologna, Italy, 2002.

[44] D. Wetherall, J. Guttag, D. Tennenhouse. *ANTS: A toolkit for building and dynamically deploying network protocols*, IEEE OPENARCH '98, April 1998.

[45] M. Wooldridge and N. Jennings, *Pitfalls of Agent-Oriented Development*, Proc. of the 2nd Int'l Conference on Autonomous Agents, 1998.

[46] B. Zenel, *A proxy based filtering mechanism for the mobile environment*, PhD Thesis, Columbia University, 1998.