ReAgents: Behavior-based Remote Agents and Their Performance

Eugene Hung
University of California, San Diego
Computer Systems Laboratory
San Diego, CA
eyhung@cs.ucsd.edu

Joseph Pasquale
University of California, San Diego
Computer Systems Laboratory
San Diego, CA
pasquale@cs.ucsd.edu

ABSTRACT

We present a performance analysis of an agent-based middleware system we have developed based on "reAgents," remotely executing agents that enhance the performance of client/server-based Internet applications. 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 application-specific logic. This is made possible through the use of *behaviors*, i.e., common patterns of actions that exploit the ability to process and communicate remotely. We model the performance of each of the primary behaviors, and provide an experimental evaluation of reAgent performance.

Categories and Subject Descriptors

D.2.11 [**Software**]: Software Architectures [Remote Agents]

General Terms

Design, Performance, Experimentation, Reliability

Keywords

remote agents, dynamic deployment, design patterns

1. INTRODUCTION

ReAgents support the remote processing of client/server application requests and responses via dynamically deployed mobile code, which carries out work tailored to the particulars of the client device. A reAgent acts as an intermediary between client and server, but maintains the client/server model (Fig. 1). As a simple example, 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 response.

Our system is designed with the following goals:

1. Provide the programmer a better way to handle the client's needs and limitations

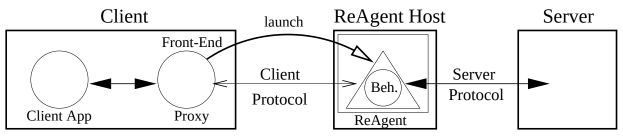

Figure 1: Client-ReAgent-Server Model

2. Be transparent to servers (i.e., do not require modifications to servers)
3. Be easy to program and use

To meet these goals, we propose a remote code mechanism that is more flexible than proxies but less complicated than fully-general mobile agents. We achieve this compromise by 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 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). These restrictions simplify the requirements of the underlying remote code execution mechanisms and reduce the potential for malicious behavior, while still deriving many of the benefits of remote execution.

A reAgent is launched to operate at a remote location because there is an advantage in doing so, such as superior computing or network resources. This location is known as the *reAgent host* (Fig. 1). The reAgent then acts as a customized dynamic proxy for that particular client, and returns the server response to it. 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.

A 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. Examples of general behaviors include filters, monitors, cachers, and collators. Once a general behavior is selected for a reAgent, it is then *specialized* for specific application needs via code parameters.

For the reAgent code to migrate to and run on the reAgent host, some middleware system that supports remote execution must be available. Our reAgent architecture does not specify its own remote execution mechanism. Instead, we leverage existing work in mobile code by translating the launch procedure into the appropriate code movement calls for each system. These movement calls

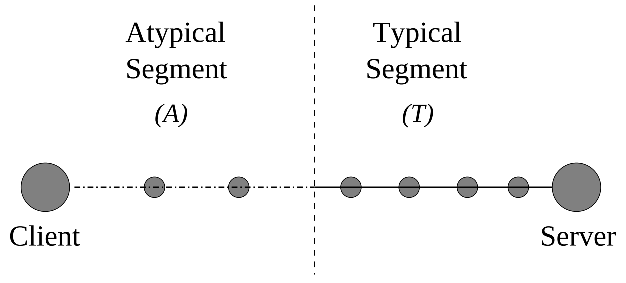

Figure 2: Operative network environment

are translated at run-time when the agent is launched, when the type of middleware on the target reAgent host is specified in a configuration file. The underlying mobile code middleware system is completely transparent to the programmer, who must only understand the simplified reAgent interface. For example, to experiment with reAgents we used a locally-developed and stand-alone Java-based mobile code system [16].

Our experience to date with reAgents has been in the context of Web-based applications [10], and mobile-computing applications [11]. In this paper, we go beyond these studies and analyze the micro-performance of each individual behavior using analytical models, and we present a set of experiments that demonstrate the macro-performance of ReAgents.

The rest of this paper is organized as follows. Section 2 presents a model of the network environment which allows us to reason about the advantages of reAgent location and execution. Section 3 analyzes the general behaviors that are the building blocks of ReAgents. Section 4 presents experimental results. Section 5 describes previous work in this area. Finally, Section 6 presents our conclusions.

2. SYSTEM ENVIRONMENT MODEL

In this section, we present a model of a system environment where an advanced remote code mechanism would be useful.

2.1 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. 2).

T consists of physical network link(s) whose levels of performance and reliability are typically what is observed in the core of 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 composite 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 generally 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 predefined protocol P_S, which we refer to as the *server protocol*.
- The client knows the above server protocol P_S.
- The client knows its network limitations and how it deviates from the server's model of a standard, powerful, well-connected client.

2.2 Regions

A *region* is an encapsulation of part of the environment that incorporates 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 developer knows its deviations (and possibly, where necessary, how to measure them). Examples of such deviations 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

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.)

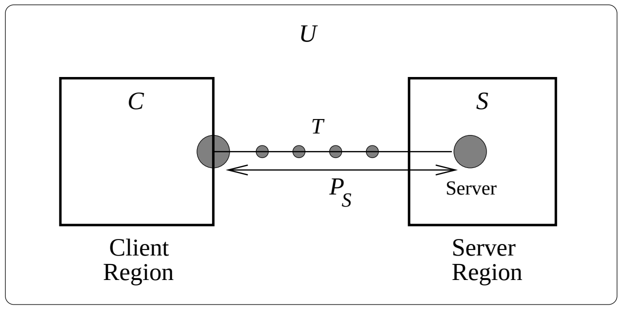

Figure 3: Server view of environment

Code in the server region operates as if the network environment were as depicted in Fig. 3. 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 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. 4), 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.

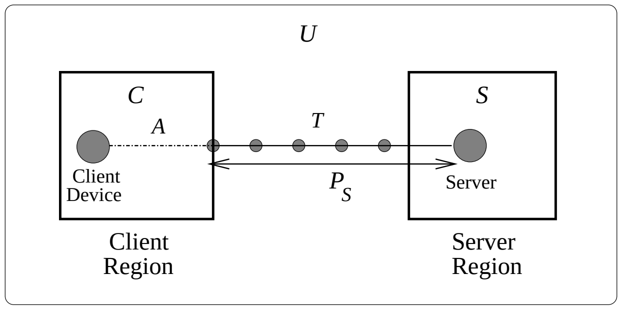

Figure 4: Client view of environment

2.3 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:

- Within the client region
- Within the server region
- Within the network somewhere in between the client and server regions

Each of these three regions 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 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, as a **reAgent**, in the client region. The reAgent is originated by the client and placed on a machine that resides in the typical segment, beyond the atypical segment. Once the reAgent 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. 5)

From the model, we can see that the reAgent, 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 reAgent 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 reAgent with an understanding of the server protocol P_S. Moreover, with the addition of the reAgent as an intermediary between client and server, the client can now communicate with the reAgent 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 reAgent.

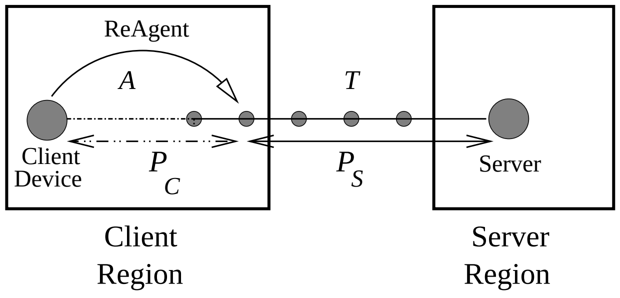

Figure 5: Bounding of Client Region

3. 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: (1) use of remote computational resources; (2) avoiding or minimizing the effects of a problematic portion of the network (high delay, low bandwidth, low reliability, etc.); (3) 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.

3.1 Filter

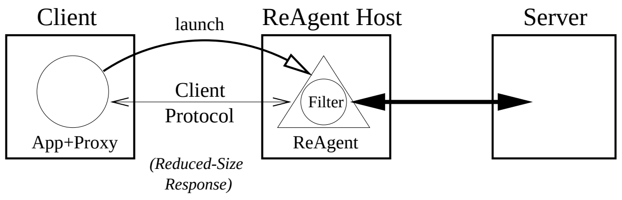

Figure 6: The Filter Behavior

Description. The Filter behavior (Fig. 6) 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. 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. 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:

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

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:

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

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.

3.2 Monitor

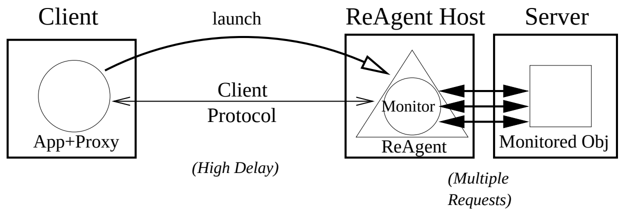

Figure 7: The Monitor Behavior

Description. The Monitor behavior (Fig. 7) 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. 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 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.

3.3 Cacher

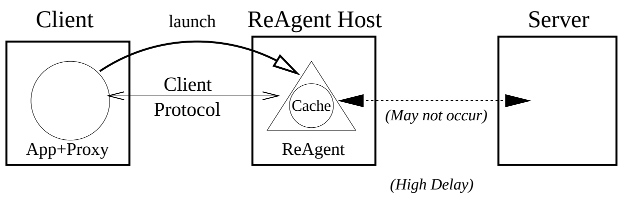

Figure 8: The Cacher Behavior

Description. The Cacher behavior (Fig. 8) 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. 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* that 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.

3.4 Collator

Description. The Collator behavior (Fig. 9) 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).

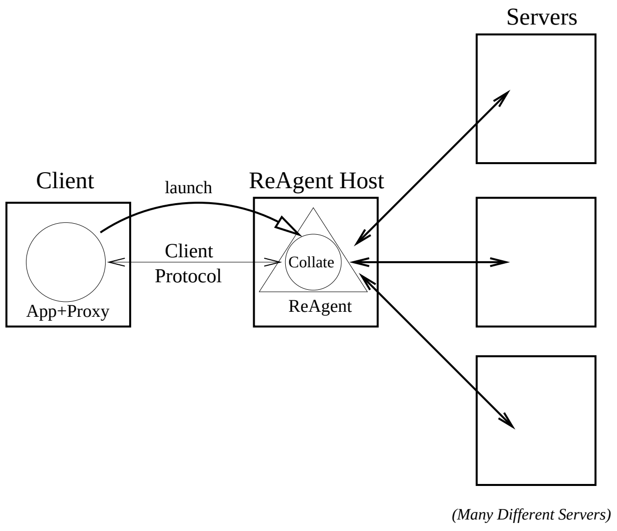

Figure 9: The Collator Behavior

Base Logic. 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. 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 waiting is completed, the server responses are collated in an application-specific way (as 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 (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.

4. EXPERIMENTS

Previous papers[10, 11] quantified the low overhead and utility of reAgents used for data compression and image filtering, based upon the Filter behavior. Here, we describe an experiment that validates the theoretical advantages of the Monitor behavior.

A primitive stock server and client application were developed. 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.

4.1 Response Time

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

1. Traditional RPC
2. ReAgent-based computing
3. Service-based customization

To introduce a non-trivial latency, the client machine, *federation*, a Sun Ultra 1, was stationed at CMU 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 1: Comparison of Response Time for Various Paradigms

	Response Time (*ms*)			95% Confidence
Paradigm	Min	Max	Mean	Interval
Traditional	71	3474	79.6	± 1.8 ms
reAgent	1	14	1.161	± 0.008 ms
Service	0	11	0.051	± 0.005 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 when 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-customized service. Of course, the cost of building applications with a similar level of performance with reAgents is much smaller than requiring all servers to 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.

4.2 Application Performance Comparison

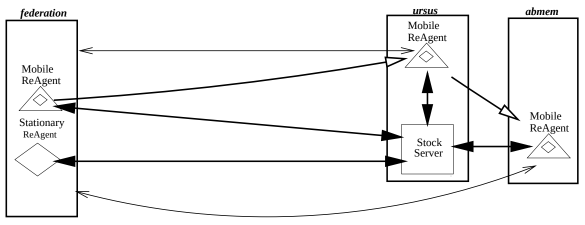

Figure 10: Stock Traders Setup

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 2.

Table 2: Mobile vs. Stationary ReAgent

Mobile reAgent Host	Mobile reAgent Success Rate	95% Confidence Interval
client	50.8%	± 1.4%
server	100.0%	± 0.0%
reAgent host	100.0%	± 0.0%

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

5. RELATED WORK

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 [23] argues for putting customizing logic at the network level. While various active networks [26, 19, 1] are able to effectively support a wide variety of client devices, they do so at the expense of deployability [6].

Another customization solution lies in the use of proxies, which act as intermediaries between client and server. While traditional proxy applications concentrate on caching and firewalls [14], some have focused on actively customizing application behavior (dynamic proxies) [4, 24, 15].

Another 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) [17] are two examples of this type of approach, where the server programmer is responsible for anticipating common client problems and catering to them. To be less dependent on the server, researchers have developed customizers with more client-side support [28, 22, 3].

A significant area of past research in client-based customization has been based upon *mobile agents* [5]. Mobile agents are pieces of customizing logic that have a persistent identity, moving around the network to multiple sites [25]. The D'Agents project [8] and IBM Aglets Workbench [13] are prominent examples from industry and academia, respectively, of systems that support the execution of mobile agents. A fuller description of these and other important agent systems can be found in [9].

Our reAgent approach is a middle-ground solution, based on a one-shot remote code mechanism that is more flexible than proxies but less complicated than fully-general mobile agents. This model is similar to that of remote evaluation [21]. 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 [12].

Finally, behavior templates are similar to the idea of "design patterns." In [27], the authors describe 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 [7]. Design patterns have previously been proposed for use in designing mobile agent applications [2, 18, 20].

6. CONCLUSION

In this paper, we described the behavior-based architecture and performance of reAgents. 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, and (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 what we believe to be 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 presented four such general behaviors — filtering, caching, monitoring, and collating — and using analytical models showed the conditions under which performance will improve when compared to the straight client/server model. We also presented empirical results from experiments that provide evidence of the performance benefits of reAgents.

7. REFERENCES

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

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

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

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

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

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

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

[8] 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.

[9] 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.

[10] E. Hung and J. Pasquale, *Web Customization Using Behavior-Based Remote Executing Agents*, Proc. 13th Int'l World Wide Web Conference, May 2004.

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

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

[13] 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.

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

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

[16] T. Newhouse and J. Pasquale, *Java Active Extensions Scalable Middleware for Performance-Isolated Remote Execution to Enhance Wireless Network Applications*, Elsevier Computer Communications Journal, Vol. 28 (14), Sept. '05.

[17] Open Mobile Alliance, *Affiliate Specifications*, http://www.openmobilealliance.org/tech/affiliates/index.html

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

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

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

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

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

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

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

[25] Y. Villate, A. Illaramendi and 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.

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

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

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