Web Customization Using Behavior-Based Remote Executing Agents

Eugene Hung
Dept. of Computer Science and Engineering
University of California, San Diego
San Diego, CA 92093
eyhung@cs.ucsd.edu

Joseph Pasquale
Dept. of Computer Science and Engineering
University of California, San Diego
San Diego, CA 92093
pasquale@cs.ucsd.edu

ABSTRACT

ReAgents are remotely executing agents that customize Web browsing for non-standard resource-limited 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 runtime network communications, and provide a general interface for programmers to more easily implement their application-specific customizing logic. This is made possible by the availability of remote *behaviors*, i.e., common patterns of actions that exploit the ability to process and communicate remotely. Examples of such behaviors are transformers, monitors, cachers, and collators. In this paper, we identify a set of useful reAgent behaviors for interacting with Web services via a standard browser, describe how to program and use reAgents, and show that the overhead of using reAgents is low and outweighed by its benefits.

Categories and Subject Descriptors

D.2.11 [**Software**]: Software Architectures [Remote Agent Behaviors]

General Terms

Design, Experimentation, Performance, Reliability

Keywords

Web customization, dynamic deployment, remote agents

1. INTRODUCTION

The trend towards smaller, wireless Internet-access devices has brought about a wide disparity in user demands, leading to problems with common applications such as Web browsing. Not only do resources and connections of client devices vary greatly, but sometimes users have radically different needs that are unaddressed by servers. And currently, little can be done for the increasingly common scenario of users with transient, resource-limited client devices entering the network unexpectedly, demanding unusual services, and finding the services unsatisfactory due to a server's inability to flexibly handle the user's situation.

To give some concrete examples of the problems of increasing client heterogeneity, consider the typical scenario of a user purchasing merchandise on the Web. If the client device has below-average computing or network resources, or the user has unusual demands, problems can arise. A client device with limited network bandwidth and display might be too slow when a server responds with extraneous images and videos of the merchandise. Another user with an unreliable connection to the Internet may be unable to verify that a transaction was completed, possibly causing the sending of a duplicate transaction order due to an intervening disconnection. A third user may require a specialized service unsupported by the server. These types of problems degrade quality of service, and will increase in frequency as more and more users, with vastly different needs and client devices, access the Web.

Our approach to addressing these types of problems is to support customization of Web browser requests and (especially) responses via dynamically deployed, remotely operating code, tailored to the particulars of the client device. For example, a Web browser running on a client device with a small, limited display will benefit from customizing logic operating at or near the server that transforms the rich server data into a spartan text-only response. Or, on a client device with an unreliable connection, browser performance would be improved from customizing logic 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 clients with unusual requests, a specialized service could be implemented as part of the customizing logic, acting as a higher-level service relative to the services provided by the server.

1.1 Previous Solutions

Server-based customization

The idea of customizing Web services is not new, but previous efforts have been divided on how and where to provide this functionality. The most common approach is to have servers adapt to the specifics of each individual client. Server-provided CGI scripts/forms, and the WAP protocol [1] are two examples of this approach, where the server programmer is responsible for anticipating common client problems and desires, and catering to them.

While easy to deploy on an individual level, server-based approaches lack generality: some servers may support a specific client while others may not. Also, even if one restricted communications only to servers that catered to one's needs, performance is unsatisfactory if the client environment changes or new clients with different needs arrive.

Intermediaries

A more scalable solution is to have the customizing logic operate as a user-level intermediary on a machine between the client and server [2]. Such an intermediary would act as a standard client

WWW2004, May 17–22, 2004, New York, New York, USA.
ACM 1-58113-844-X/04/0005.

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, transform 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, 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* [3]. By a mobile agent, we simply mean code that is capable of migrating from the client to a remote site, acting on behalf of the client. The most general forms of mobile agents, which allow suspension during execution and consequent migration, are extremely flexible and powerful in their support for customization. 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. However, mobile agents typically require complex underlying middleware systems to handle the semantics and security problems that are a byproduct of code migration. They are also not easy to program, as programmers are generally not familiar with the mobile code programming paradigm.

1.2 Our Solution: reAgents

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

- provide the user 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 customization 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, which we call 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 once, and does so *before* it begins execution. This is based on the extended client/server model in [2] and similar to the remote evaluation model in [4].

A reAgent is customized and launched by the user 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. 1). The reAgent is then connected to the browser through a front-end proxy interface that intercepts browser requests and forwards them to the reAgent for handling. The reAgent then acts as a customized proxy for that particular user, and returns the server response to the front-end proxy, which forwards the response to the browser. This extends the capabilities of the browser 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 browser 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 [5].

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. 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. As perhaps the simplest and most common example, the "transformer" behavior, which in general processes a response from a server before passing it on to the client, can be specialized in terms of how the server data will be transformed. A client device with a limited-color display would benefit from a data transformation that reduces the color depth of image files, while another client device with a small display would benefit from a data transformation that shrinks images.

Some of the useful behaviors we have identified are the following:

- *transforming*, by changing the form of a server response before it is communicated to the client, as in filtering the response before sending over low-bandwidth links to reduce bandwidth and latency, 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 near the client to improve responsiveness when there is high network latency between client and 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 are to be forwarded to numerous servers, to a more efficient operating point 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).

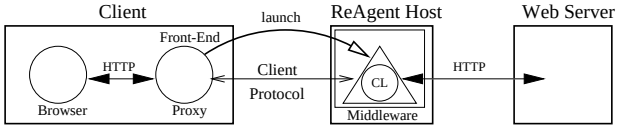

Figure 1: Client-ReAgent-Server Model and Environment

By identifying the characteristic behaviors of reAgents and using them as the building blocks for development, we provide a simple, structured way of building and deploying reAgents that efficiently customize server data in a client-specific, scalable fashion.

The rest of this paper is organized as follows:

In Section 2, we detail the concepts behind reAgents and their composing behaviors. We also catalog the behaviors we have identified. In Section 3, we describe the programming interface for the external components of the reAgent system. In Section 4, we give an example of an application that uses the reAgent system to create a powerful customized stock trader. In Section 5, we present experimental results that show that the implementation overhead of this approach is tolerable and outweighed by its benefits. In Section 6, we review previous work in the area of Internet application customization. Finally, in Section 7, we present conclusions.

2. REAGENTS AND BEHAVIORS

2.1 ReAgent Environment

ReAgents operate in an environment with somewhat inflexible (i.e., difficult to change) Web servers and browsers. Both server and browser must be linked to the reAgent, which is interposed in their communications. The reAgent communicates with a 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 browser request to the server without changing the internal browser code. This is done simply by using a proxy service (the *front-end proxy*) that runs on or near the client device. The browser's output is redirected to the front-end proxy's input, and then the front-end proxy forwards the output to the reAgent. It 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 can be set up between the front-end proxy and the reAgent, as well as between the front-end proxy and browser (the browser'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 browser) to the reAgent, and acts a server to the browser by passing responses from the reAgent to the browser. In this manner, the implementation of the browser and server remains unchanged.

For the reAgent code to migrate to and run on the reAgent host, some form of middleware system that supports remote execution must be available. The internal implementation of reAgents is not tied down to a specific format of remote execution, which would limit its scope. Instead, it leverages existing work in mobile code by translating the launch command into the appropriate movement methods for each system. This translation occurs at run-time when the agent is launched, with the type of middleware on the target reAgent host specified in a configuration file.

2.2 Model of Operation

A reAgent (Fig. 2) is composed of specialized behaviors, which in turn are instantiations of a general behavior. Behaviors are patterns of remote action and communication (with a server). As a programming object, a behavior begins in the form of a template, consisting of core logic that captures the general actions of the behavior, and an API that allows it to be specialized with programmer-provided customizing logic and to be controlled by the user during run-time (control methods).

A behavior operates as follows. As part of the reAgent, it waits until it receives a request. This request is passed as an input to the core logic, which may call some methods implemented by the client (the customizing logic, or CL). At some point during the core logic operation, request(s) are made to a server, which returns a response. This response is also passed through the core logic, and possibly some more CL methods, before being output. At no point does the behavior *initiate* communication with the client; the reAgent containing the behavior is responsible for that, as well as for handling the input and output. During execution, the client can tell the reAgent to invoke the control methods of the behavior, which help externally control the reAgent's behavior.

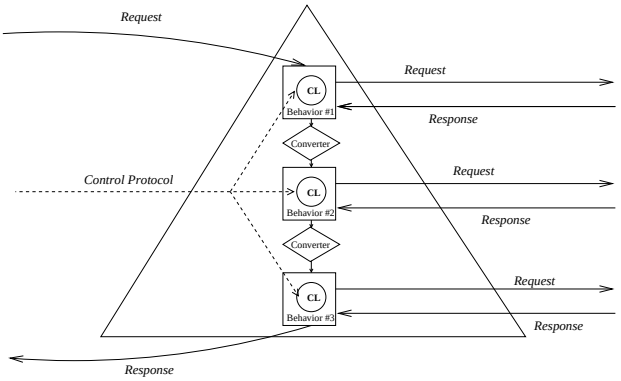

Figure 2: ReAgent Architecture

Because the input and output are handled at a higher level in a consistent fashion, multiple behaviors can be combined in a chain. What is needed is a *converter*, 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, get the output, feed this output to the converter, get the converter's output, and feed the converter's output to the next behavior in the chain.

Finally, a reAgent and its behaviors are able to customize communication. For each behavior, a server request is made using the *server protocol*. The server protocol is defined as the protocol that the server understands (in the case of the Web, HTTP). Meanwhile, the reAgent is able to communicate with the client via a two-tier protocol, the top layer being the *control protocol* and the bottom layer being the *client protocol*. The control protocol is fixed by implementation and is the means by which the client can invoke the behavior control methods. 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. A useful scenario for a custom client protocol comes when browsing the Web, where a portion of the network path near the client is relatively unreliable, e.g., wireless access. One could launch a reAgent to a location past the unreliable portion, e.g., beyond the wireless base station, set up a more stable, reliable protocol between the client and reAgent, while continuing to use HTTP between the reAgent and the Web server.

2.3 Behavior Library

Our focus is on behaviors useful for composing reAgents. A behavior is "useful" if it intrinsically exhibits benefits derived from a reAgent's ability to operate remotely. These benefits come from some combination of, but are not limited to, the following:

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

The following sections catalog the useful behaviors we have identified. (This catalog is not meant to be exhaustive, but exemplary.) For each behavior, we present a description of the behavior, an outline of the core logic, and a common Web application using the behavior. To simplify the exposition, we show the core logic in pseudo-code, and describe what happens when the behavior is ordered by the reAgent to process a request.

2.3.1 Transformer

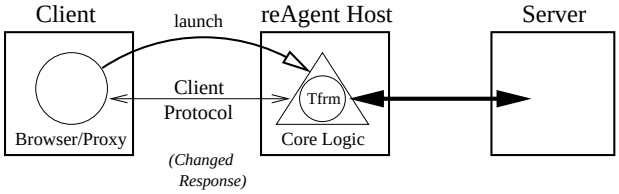

Figure 3: The Transformer Behavior

Description. The Transformer behavior (Fig. 3) is used whenever there is a need to modify a response sent from a server to the client. The CL (customizing logic) is the application-specific algorithm that defines *how* to change (transform) the data.

Core Logic

```
input: request

-----------------------------------------------------------------
serverProtocol.send(request)
response = serverProtocol.receive()
newResponse = transformerLogic.transform(response, args)
-----------------------------------------------------------------

output: newResponse
```

The behavior passes the request through to the server, runs the transforming algorithm on the server's response, and sends the modified response back to the client.

Application. The Transformer behavior is designed for scenarios where the server data is in an unsatisfactory format for the client. One frequently occurring 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 transforming the data by filtering extraneous or unusable data before sending it to the browser would conserve bandwidth without significantly impacting the perceived quality of the data.

As an aside, a trivial but useful form of this behavior is a "null Transformer," where the CL simply outputs whatever data is input, untouched. This captures the behavior of *relaying* data and is useful when used in combination with other behaviors that perform an action and then need to communicate briefly with the server.

2.3.2 Monitor

Description. The Monitor behavior (Fig. 4) is designed for use in applications that have a need to frequently observe the state of a remote object (on a distant server) until a certain state is reached (determined by the response). The calculation of the next attempt to observe, plus the response evaluation function, forms the CL.

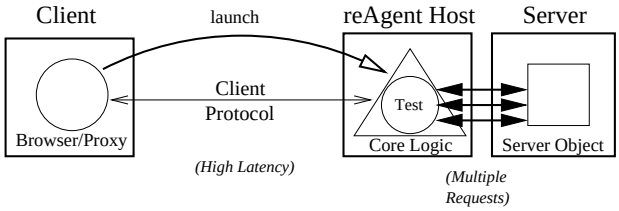

Figure 4: The Monitor Behavior

Core 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*, a function of the response, has been reached. Once this occurs, monitoring is terminated and the response corresponding to the trigger state is returned.

Application. A simple example of a Web application for a Monitor involves intelligent auto-refresh. 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 if the data remains unchanged, saving bandwidth.

2.3.3 Cacher

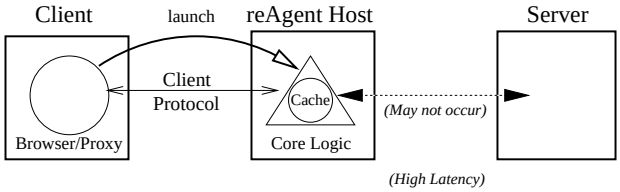

Figure 5: The Cacher Behavior

Description. The Cacher behavior (Fig. 5) 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. This behavior is especially useful for applications that have frequent but similar requests to remote servers.

Core 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 customizing 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 associates the request with a "key" (derived from the protocol) and 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 occurs whenever the server sends the behavior a response. 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 when the client request key matches a key in 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 browsers support some form of caching. Without intermediate hosts, the server data is stored on the client device. While this practice is optimal for minimizing network latency, some client devices have limited amounts of memory (or CPU power) such that cache performance is seriously degraded by running locally. These resource-poor clients would benefit from moving the cache to a nearby location with sufficient resources.

2.3.4 Collator

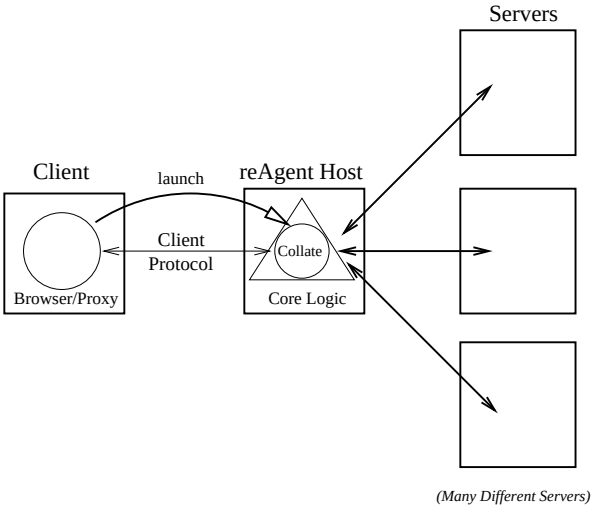

Figure 6: The Collator Behavior

Description. The Collator behavior (Fig. 6) transmits the same request to multiple servers from a remote location, and waits until a wait condition, specified by the client, is met. Afterwards, the responses are sent to an application-specific function that produces a result for the client (collating).

Core 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
```

A request is sent once to the reAgent, which then transmits copies of it to each server. The reAgent then waits for responses from the servers in an application-specific fashion, as defined by the `wait()` method. 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 all servers within a timeout period. After the `wait()` method returns, the server responses are collated by the `collate()` function, 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 besides price, such as delivery time or seller reputation.

3. SOFTWARE INTERFACE

3.1 Programming Language

We chose to use Java [6] as our implementation language for reAgents for several reasons. First, Java's modularity allows the delineation of the CL from the core behavior logic as separate *classes*, or method packages. Second, Java's dynamic extensibility allows the pattern code to instantiate and load classes on demand, a necessity for flexible specialization of behaviors and converters. Java programs are also portable, only requiring that the destination server be able to run a Java Virtual Machine (JVM). Portability is highly advantageous in any heterogeneous environment. Finally, JVMs are currently a *de facto* standard for running foreign code on the Web, making Java-based applications highly deployable.

We expect that any programmer wishing to employ reAgents in their applications will use mobile agent middleware that is both able to invoke Java programs, and have their movement and communication procedures be invoked in return by a Java program. We also require that the middleware be able to obtain a copy of the CL, whether it be carried by the agent or downloaded from a software repository. These requirements are all met if the agent system is written to support Java-based agents. Many if not most mobile agent systems (such as Aglets [7] and D'Agents [8]) support Java-based agents.

3.2 Application Programming Interface

The following sections define the external interfaces for the user-defined components of the reAgent: the communication protocols, the behaviors, the converters, and the reAgent itself. Any customization performed by the reAgent must use at least one, if not all, of these interfaces.

3.2.1 Custom Protocol Interface

All protocols used by the reAgent must conform to a `Protocol` interface:

```
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 interface defines 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 the client in a customized fashion.

The `Protocol` interface enables reAgents to support standard protocols such as HTTP or TCP. Library implementations of these protocols are provided to the reAgent programmer to simplify the programming in standard cases.

More generally, this is also where the programmer can implement any custom code that requires execution at both the client and the reAgent, such as compression/decompression and encryption/decryption.

3.2.2 Behavior Interfaces

The behavior interface is different for each behavior type, as each behavior supports different methods. However, all behaviors have a few methods in common:

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

  public void pause ();
  public void resume ();
}
```

- `Behavior()` is the constructor, which constructs a specialized behavior. A programmer must specify the type of behavior, the customizing logic (the CL), the logic-specific arguments to the CL, and the implementation of the server protocol. These are provided as parameters to the general implementation of the core logic (which acts as a template for the behavior):
- `pause()` is a control method that causes all CLs in the behavior to be ignored. The request and response are still made, as if the specializing behavior did not exist.
- `resume()` is a control method that only has an effect on a paused behavior. The behavior is unpaused and its CLs become operational on the requests and responses sent to it.

The result is a specialized behavior that acts similarly to other behaviors created from the same type, but differs in a specific, custom manner (by the CL, the arguments of the CL, or the server protocol).

The following section describes both the CL interface and the control method interfaces for each of the behavior types.

Transformer

```
public interface Transformer extends Behavior {

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

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

Customizing Logic method:

- `transform()` transforms the server data in a client-specific fashion.

Control method:

- `setLevel()` changes the level, or amount of transformation to the amount specified in the argument. (The level is a class variable and is parsed according to the CL.)

Monitor

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

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

Customizing Logic methods:

- `calcNextQuery()` returns the next time the monitor should make another query.
- `testResponse()` tests to see if the server response has produced a trigger state.

Control method:

- `sendQueryNow()` forces an immediate query to the server.

Cacher

```
public interface Cacher 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 Logic methods:

- `hash()` takes a request as input and returns a `String` that is the key string for that request.
- `lookup()` returns whether the key string is in the cache.
- `get()` returns the `Response` associated with a key string in the cache.
- `replace()` puts a key string in the cache and associates it with a `Response`. This method also implements the cache replacement policy.

Control methods:

- `flush()` removes all entries from the cache.
- `changeCacheSize()` changes the size of the cache.

Collator

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

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

Customizing Logic methods:

- `wait()` defines how long the reAgent should wait for server responses.
- `collate()` takes all the results received and combines them into one object to be sent back to the client.

Control method:

- `forceCollate()` interrupts the `wait()` method and forces the `collate()` method to be called immediately.

3.2.3 Converter Interface

The reAgent programmer who chains behaviors together is responsible for writing the conversion algorithm so that the output of the first behavior can be parsed as input to the second behavior. The interface of a converter is simple:

```
public interface Converter {
  // convert function
  byte[] convert (byte [] content, String [] args) throws IOException;
}
```

where `convert()` converts an input response into an output request to be used as input for the next behavior.

3.2.4 ReAgent Interface

The `reAgent` interface is used to create and control reAgents. These methods are called from the front-end proxy.

The reAgent interface is :

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

  public boolean launch (String hostname, String configFile);
  public void addBehavior (Behavior behavior, Converter converter);
  public Object process (Object request);
  public void stop ();

  public void sendControlMessage (String message);

}
```

where:

- `ReAgent()` creates an object that, after migration, will communicate with the client with the protocol specified as a parameter.
- `addBehavior()` adds a specialized behavior object (Section 3.2.2) to the internal reAgent queue. If the behavior is to be chained to a previous behavior, the converter (Section 3.2.3) that will change the output of the previous behavior into acceptable input is also added.
- `launch()` launches the reAgent to a remote site, specified as the first parameter. The second parameter, the configuration file, defines system variables that enable the reAgent to migrate. For example, a configuration file specifies the type of agent system that the reAgent host is running so that the correct migration methods are called.
- `process()` sends a request to the launched reAgent for processing.
- `stop()` stops the reAgent and begins the process of removing it from the reAgent host.
- `sendControlMessage()` allows the client to send a message to the reAgent using a pre-defined control protocol. (The control protocol is defined by the messaging system of the middleware running the reAgent.) Such a message can be used to exert fine-grained control over an individual behavior.

4. USAGE

This section describes how a Java-based implementation of reAgents is used to simplify customized extension of Web browsers. First we describe how a reAgent is used in conjunction with a browser with access to a Java Virtual Machine. Then, we walk through an example that uses a reAgent with two behaviors, monitoring and transforming, to customize stock trading beyond simple limit orders.

4.1 Building and Using a ReAgent

The Java implementation of reAgents is a code package that implements the general reAgent behaviors for existing mobile code systems. To build a reAgent, its behaviors and converter components need to be specialized beforehand by calling the `Behavior` and `Converter` constructors with the appropriate client-specific arguments. Then, a reAgent object is constructed with a custom client protocol, and each of the desired components are added to the reAgent via calls to the `addBehavior` method. This can be done directly using the front-end proxy code, or, for ease of use, through a GUI such as a web form. In the latter scenario, the form calls a script that creates all the Behaviors, Converters, and Protocols in the form, and then links them together by constructing a reAgent object calling the reAgent constructor and `addBehavior` methods to create a reAgent object that is usable by the front-end proxy. If a reAgent consists of more than one behavior component, the behavior inputs and outputs are chained together in the order they are added to the reAgent.

After the reAgent is constructed, it is launched to the reAgent host by calling the `launch` method. This can be done via a Web form or code. The reAgent is sent to the reAgent host, and the client and reAgent protocols are set up for communication.

After this launch, the user redirects the browser to use the front-end proxy linked with the reAgent so that all of the browser's requests will be intercepted by the reAgent. In this manner, the browser no longer communicates with the server; it only communicates with the reAgent (using the client protocol of the front-end proxy).

When the reAgent is launched via a Web form, a browser control window is automatically made available. The control window gives the user the means to dynamically control the operation of the reAgent by giving the user the ability to call the control methods of the reAgent's behaviors at run-time. Actions here result in messages sent by the front-end proxy to the reAgent via the control protocol that invoke the corresponding control methods of the behaviors in the reAgent. Only control methods corresponding to the behaviors composing the reAgent appear in this control window.

When the user no longer wishes to use the reAgent, the proxy settings can be changed to no longer communicate with the reAgent front-end proxy, or the control window can be closed to send a stop message to all the behaviors in the reAgent.

4.2 Application: Custom Stock Trading

Many investors are now using the Web to manage their money. With large amounts of money at stake, minimizing response time to client requests is important. While server programmers can anticipate common client requests, such as executing a stock transaction as soon a certain price is reached (a limit order), more unusual but desirable requests may not be supported. For example, many investors believe it is important to buy or sell after the stock price crosses the 200-day moving average, or after the stock price reaches a new local extrema for the third time in an arbitrary time period. Thus, a window of opportunity may be missed if the server does not offer the ability to transact after a complex formula is satisfied.

To address this problem, the user can customize a reAgent to migrate close to the stock server, repeatedly query the stock price from a nearby location, evaluate the current stock price based on its custom algorithm, and send a buy or sell order once that custom algorithm recommends action. This involves the combination of two behaviors: the Monitor, which monitors stock prices until some condition is satisfied, and a null Transformer (which has an empty CL) to perform a simple request/response (to buy or sell).

The user can build this reAgent by first developing the custom stock-evaluation algorithm and coupling it with the Monitor behavior's API to create a class file that implements this algorithm, viz. `CustomStockMonitor.class`. The user must also create a null Transformer. These behaviors are created with the following method calls:

```
Behavior stockMonitor = new Behavior("Monitor",                  // type
                                     new HTTP(),                 // protocol
                                     "CustomStockMonitor.class", // CL
                                     null)                       // CL args
Behavior purchaser    = new Behavior("Transformer",              // type
                                     new HTTP(),                 // protocol
                                     null,                       // CL
                                     null)                       // CL args
```

The user also needs to develop a class, using the Converter API, that translates a server response into a BUY request, and puts it in a class file, viz., `BuyStock.class`.

```
public class BuyStock implements Converter {

  public byte[] convert (byte[] input, String[] args) {

     // parse input for stock ticker name and return request to BUY stock

     // this is a private custom function (not shown here, as it
     // varies based on the server output) that parses the HTML
     // response for the ticker name
     String stockName = getTickerName (input, args);

     String response =
       "GET http://www.stockbroker.com/buy.cgi?ticker=" + stockName;

     return response.toByteArray();
  }
}
```

The user (via HTML form or code) then has the front-end proxy create a reAgent with a standard HTTP client protocol, the two behaviors instantiated above, and an instance of the class `BuyStock`.

```
reAgent = new ReAgent(new HTTP());
reAgent.addBehavior (stockMonitor, null);
reAgent.addBehavior (purchaser, new BuyStock());
```

The reAgent is launched to the reAgent host `middleman.org` with the code :

```
reAgentHost = "middleman.org";
reAgent.launch (reAgentHost, configFile);
```

Upon launching, the reAgent will wait for a request from the client. Say the client requests that the reAgent monitor the stock price of IBM:

```
request = "GET http://www.stockbroker.com/quotes/?p=IBM"
reAgent.process (request);
```

The reAgent executes the Monitor behavior at the remote host, and the specialized algorithms in `CustomStockMonitor` are called to determine how often to request a stock quote, and whether the situation is attractive enough to buy IBM. When the algorithm issues a buy signal, the Monitor returns the HTML response that triggered the buy signal to the reAgent. The reAgent then feeds this response to the `BuyStock` Converter, which turns it into a BUY request for the appropriate stock, which in turn is fed into the Transformer component. The server then receives the BUY request from the Transformer, executes the transaction, and sends a response confirming the transaction back to the Transformer. The Transformer, with a null CL, returns it unchanged to the reAgent, which returns it to the client. All of this complexity is handled automatically by the reAgent.

After launching, the user has the option to control the reAgent via the control window. This provides the user with the ability to pause or resume the reAgent, and the Monitor behavior also gives the user the ability to override it by forcing a query at a specific time.

5. EXPERIMENT

We have implemented the behaviors in Java, on top of a locally-developed Java mobile code system called Java Active Extensions (JAE), a middleware system that supports one-shot code mobility [9]. To experimentally evaluate the overhead introduced by reAgents, we implemented a simple filtering example, based on the Transformer behavior. We show that the overhead is low, especially when taken relative to the performance gains derived by a compressing reAgent.

5.1 Experiment: Image Reduction

In this experiment, the transformer was used to simply reduce 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 Images package.

5.1.1 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 due to 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, transformation), and sending it to the client.

5.1.2 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 found on the Web, 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.

5.1.3 Results

The results, compared to a non-transforming Web browser, are summarized in Fig. 7. For most of the files, the transforming 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 10 KB file, where the gain from compressing the data sent over the limited bandwidth link did not compensate for the reAgent processing overhead. However, the transformer provided superior performance for files greater than 10 KB. An obvious optimization would be for the reAgent to not transform small files, as the benefit does not outweigh the cost.

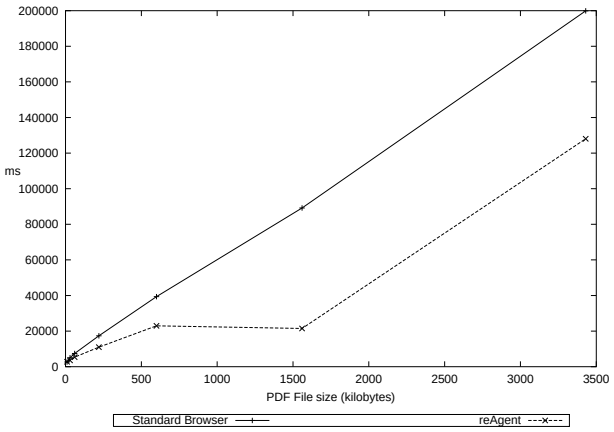

Figure 7: End-to-end comparison times

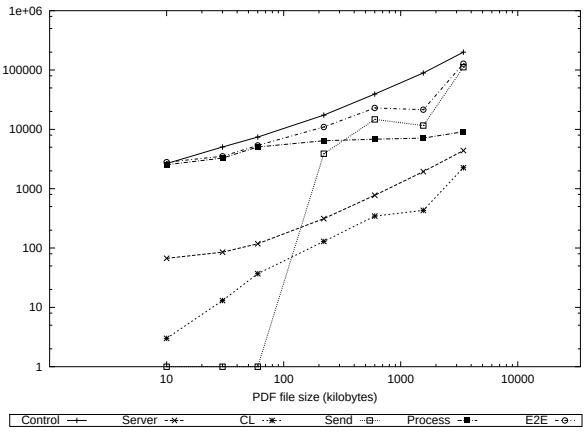

Figure 8: Overhead of transforming (log-based)

We timed different parts of the reAgent while transforming 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. 8, 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 transforming, 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 transforming 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 Transformer behavior not only imposes little overhead, but can provide significant improvements in performance over a traditional client/server application.

6. RELATED WORK

Research on customizing applications for improved performance has seen a variety of solutions. Active networks, dynamic proxies, mobile agents are but a few of the approaches advanced to solve this problem. In this section, we describe the related research in this area.

6.1 Dynamic Proxies

One type of 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. While traditional proxy applications concentrate on caching Web results for improved performance and for controlling Internet access through firewalls [10], there has been some work done on proxies that actively customize application behavior (dynamic proxies). In [11], the idea of an Active Cache, or a dynamic proxy that acts to help improve caching for dynamic Web objects, was 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, [12] 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, do not fit our goal for a scalable customization solution. New clients with correspondingly different demands are continually being created, and requiring server programmers to cater to every potential variation in client capabilities is problematic.

6.2 Client-based Customizers

For improved scalability, we examine customizers with more client-side support. In [13], the author 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. 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 [14], which describes the implementation of a middleware architecture that supports adaptive Web-based proxies called Customizers. Customizers tend to be 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. It is optimized for use over an HTTP client/server connection and not a more generic client/server connection. Finally, the Active Names project [15] 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.

6.3 Mobile Agents

A significant area of past research in client-based customization has been based upon *mobile agents*. The IBM Aglets Workbench [7] and the D'Agents project [8], 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 can be found in [16].

Mobile agents provide a robust solution for addressing the problems of client heterogeneity. And yet, mobile agent-based applications are rare. This is not due to lack of theoretical value: [17], [18], and [19] describe applications which take advantage of mobile agents. Yet, most application programmers are still unfamiliar, or have difficulty, with the mobile agent programming model. Our work differs from previous mobile agent literature by concentrating on a method that reduces the complexity of building agent-based applications.

7. CONCLUSION

We described a means for developing remotely executing agents (reAgents) that allow Web browsers to be customized to derive performance benefits for heterogeneous clients that are resource limited. The approach is based on identifying useful remote behaviors that abstract away many of the complexities of mobile code systems. When a developer uses these behaviors as a foundation for development, extending Web browsing becomes easier to implement due to pre-coded support for the movement, communications, and general processing functions used by that application's characteristic behavior.

Our main conclusions 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 set of behaviors that capture common and useful patterns of action by remotely executing agents. These behaviors are: Transformer, Monitor, Cacher, and Collator.

- We can more easily build intermediary-based applications through these behaviors. They allow the programmer to plug in 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.

8. REFERENCES

[1] WAP Forum. *Wireless Application Protocol 2.0 Specification*, http://www.wapforum.org, July 2001.

[2] J. Pasquale, E. Hung, T. Newhouse, J. Steinberg, and N. S. Ramabhadran, *Improving Wireless Access to the Internet By Extending the Client/Server Model* Proc. European Wireless, Florence, Italy, Feb. 2002.

[3] 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.

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

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

[6] K. Arnold and J. Gosling, *The Java Programming Language*, Addison-Wesley, Reading, MA, 2nd ed., 1998.

[7] D. Lange, M. Oshima, G. Karjoth, and K. Kosaka, *Aglets: Programming Mobile Agents in Java*, Proc. Worldwide Computing and its Applications (WWCA'97), Lecture Notes in Computer Science, Vol. 1274, 1997.

[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] T. Newhouse and J. Pasquale, *Java Active Extensions: A Middleware System for Remote Execution*, submitted for publication, 2004.

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

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

[12] 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), Sep. 2002.

[13] B. Zenel, *A Proxy Based Filtering Mechanism for the Mobile Environment*, PhD Thesis, Columbia University, 1998.

[14] J. Steinberg and J. Pasquale, *A Web Middleware Architecture for Dynamic Customization of Content for Wireless Clients*, Proc. 11th Int'l WWW Conf., Honolulu, HI, May 2002.

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

[16] 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.

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

[18] R. Gray, D. Kotz, et al., *Mobile Agents for Mobile Computing*, Proc. 2nd Aizu Int'l Symp. Parallel Algorithms / Architectures Synthesis, Fukushima, Japan, Mar. 1997.

[19] 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.