Using Behavior Templates To Design Remotely Executing Agents for Wireless Clients

Eugene Hung
Department of Computer Science and Engineering
University of California, San Diego
San Diego, California 92093–0114
E-mail: eyhung@cs.ucsd.edu

Joseph Pasquale
Department of Computer Science and Engineering
University of California, San Diego
San Diego, California 92093–0114
E-mail: pasquale@cs.ucsd.edu

***Abstract*— *ReAgents* are remotely executing agents derived from *behavior templates* that support wireless clients in Internet applications. 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. Behavior templates simplify the programming of reAgents by transparently handling data migration for remote execution, supporting custom communication protocols between the client and agent, and providing a general interface for programmers to implement their application-specific customizing logic. This simplification is made possible by the identification of characteristic *behaviors*, i.e., common patterns of actions that exploit the ability to process and communicate remotely. Examples of such behaviors are filtering, encoding/decoding, monitoring, caching, and distribution/collation. In this paper, we identify and analyze a set of core characteristic behaviors, describe how to program reAgents using behavior templates, and show that the overhead of using reAgents is low and outweighed by its benefits.**

I. Introduction

The trend towards smaller, wireless Internet-access devices has brought about a wide disparity in the resources and connections of client devices (Fig. 1). This leads to greater complexity in the design of Internet applications, as servers must now handle a broad range of computing power and/or connectivity quality. And currently, there is little that can be done for the worst-case scenario of mobile client devices (*mobile clients*) popping into the network unexpectedly, demanding services, and finding the services unsatisfactory due to the server's inability to flexibly deal with the shortcomings of the mobile device.

Consider the typical client/server-based electronic-commerce application that enables a user to purchase merchandise over the Internet. A typical mobile client adds many challenges that must be met. For example, a palmtop with a small display area might end up downloading images of the merchandise too big for it to display. A notebook with an unreliable wireless connection to the Internet may be unable to verify that a purchase was completed, possibly sending a duplicate purchase order due to an intervening disconnection and buying the same product twice. A wireless laptop with a strong but insecure connection (common to many wireless networks) may require levels of security beyond the server's ability to supply. These problems degrade quality of service, and will increase in frequency as mobile clients grow more diverse in their needs and resources.

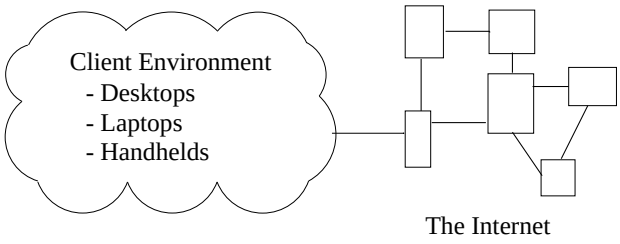

Fig. 1. A heterogeneous client environment

To address these types of problems, applications can be designed to compensate for a mobile client's shortcomings by customizing their performance for each individual client through the use of remotely-operating customizing logic. For example, an application running on a mobile client with a tiny display would benefit from customizing logic operating at or near the server that shrinks images to a size that can fit on the display. On a mobile client with an unreliable connection, the same application would use customizing logic operating at the boundaries of the connection to stabilize the connection, thus operating as a custom protocol. Finally, over a connection susceptible to malicious eavesdroppers, the application could encrypt the data according to its customizing logic before sending it to the mobile client to be decrypted.

A. Previous Solutions

The idea of providing customizing logic to client applications is not new, but previous efforts have been divided on how and where to provide this functionality. The active networks approach [1] is to place the logic "inside the network." Another approach is to have servers adapt to the specifics of each individual client, possibly through a standardized protocol such as WAP [2]. These approaches can have deployability or scalability problems. Placing customizing logic in the network (i.e., routers) is difficult and can negatively impact other network applications, while relying on servers to customize for mobile clients does not scale well, as the constant introduction of new mobile devices creates a correspondingly large number of demands that must be met.

Another approach is to have the customizing logic operate as a user-level intermediary on a machine between the client and server. Such an intermediary would act as a standard client as viewed by the server by communicating with it using the pre-established client/server protocol. The intermediary would also act as a specialized server for the client, with the ability to, for example, filter data received from the server into a reduced form for a client connected via a low-bandwidth, wireless network.

A popular type of intermediary, for which there is much research and experience, is the *proxy*. A proxy is typically 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 excellent 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 basestation 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.

B. Our Solution

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

- provide a mobile client a better way to deal with its limitations
- not affect the server
- be easy to program

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 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 similar to the remote evaluation model [4].

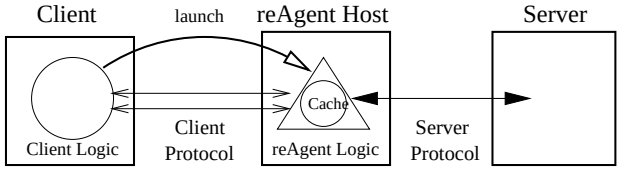

Fig. 2. reAgent Architecture

A reAgent can be viewed as an *extension* of the client, which launches it to operate at a remote location that is superior in, for example, available computing or network resources. Upon execution, the reAgent acts as a typical client to the server, while presenting a server interface to the client. The reAgent operates between client and server on a machine designated as the *reAgent host*. (Fig. 2)

One-shot migration simplifies the infrastructural support by avoiding security issues introduced by code that can roam from site to site, and avoiding the technical problems associated with maintaining and updating program state during migration, without losing much functionality, a view supported by [5]. With the bar thus lowered for hosts to support reAgents, we assume that the infrastructure in support of reAgent deployment will not be problematic. ReAgent hosts can be third-party servers charging reAgents for compute time, or personal machines that are more powerful than the client device (i.e. a user's home/office desktop can be set up to support reAgents launched from the user's roaming handheld device.)

The simplification advantages of one-shot migration also extend to development. ReAgent code can be derived from a library of templates which capture common useful forms of client/agent/server interactions; and via code parameters, can be specialized for particular application needs. By having restricted and simplified the form of movement of reAgents, we have been able to identify useful patterns of processing and communication. As this is the central contribution of our work, we now elaborate on the use of templates that codify general patterns of behavior.

Our thesis is that there are certain characteristic "behaviors" which intrinsically exhibit benefits that are derived from a reAgent's ability to operate remotely. These benefits come from some combination of, but not limited to, the following:

- use of remote computational resources;
- acting on behalf of a client beyond a problematic portion of the network;
- communicating with a server from a more advantageous location (e.g. shorter latency, high bandwidth, greater stability, etc.)

Some examples include:

- *filtering*, by removing unusable or unwanted data before it is communicated over low-bandwidth wireless links to reduce bandwidth and latency, or before it is received to reduce client storage and processing;
- *encoding/decoding*, to derive some benefit by transforming data to be communicated over a problematic link, such as improving security via encryp-

tion/decryption, reducing latency and bandwidth via compression/decompression, or improving reliability via redundancy coding);
- *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 closer to the client to improve responsiveness when there is high network latency between the client and the server, and the client does not have sufficient system resources to efficiently operate a local cache;
- *distributing/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 results can then be collated or fused before passing them back.

These example behaviors are general patterns of action that we have encapsulated in *behavior templates*, which are used to dynamically create and launch reAgents with minimal effort by the programmer. The parameters to the templates specialize the behaviors of the reAgents they generate. By identifying the characteristic behaviors of reAgents and using them as the building blocks for development, we provide a simple way (via templates) of building and deploying agents that efficiently customize server data in a client-specific, scalable fashion.

The rest of this paper is organized as follows.

In Section II, we illustrate the behavior template concept by providing some simple examples of how a programmer would use them. In Section III, we describe the set of core characteristic behaviors we have identified. In Section IV, we present experiments that show that the implementation overhead of this approach is tolerable and outweighed by its benefits. In Section V, we review related work in the area of Internet application customization in more detail. Finally, in Section VI, we present conclusions.

II. Programming Examples

To provide some intuition as to how behavior templates work and simplify programming, we present some examples in this section. The first example shows how a programmer would incorporate application-specific customizing logic that does compression/decompression with an encoder/decoder behavior template implementation written in Java. While the concepts behind behavior templates are language-independent, the choice of language for actual implementation does affect deployability. Java was chosen because of its portability — its run-time environment, the Java Virtual Machine (JVM), provides a standard, homogeneous environment for execution that allows code to be compatible across platforms. Also, Java's support for dynamic code loading and serialization of objects (to implement mobile code) across the network [6] makes it popular for mobile code systems, and allows us to leverage existing mobile code system technology.

A. Example 1: Basic Data Compression reAgent

Consider the problem of Web browsing on a laptop over a low-bandwidth connection to the Internet. Given that images can take a long time to download because of the limited bandwidth, it would be beneficial to use a reAgent at an intermediate site to compress image data before it is sent over the low-bandwidth connection. When the data is received at the laptop, it should be automatically decompressed and presented to the browser.

1) Using the Templates Package: To support this common scenario, the programmer selects the *Encoder* behavior template from the library containing the core set. This is because compression and decompression share the same defining characteristic of all encoders/decoders — they transform the server data to a different format (e.g., for performance or security reasons), send it, and then restore to the original format. The application-specific customizing logic, or CL, must also be designed. For this particular example, the programmer decides to base the CL on the popular ZLIB compression method [7], for which there are publicly available Java implementations, and which can easily be placed in the form required by our system.

2) Using the Encoder Logic Interface: The programmer must port the CL to the behavior-specific interface (in this case, the `EncoderLogic` interface in our Java implementation shown below) so that the Encoder template knows how to invoke the CL.

The `EncoderLogic` interface requires the CL to support the following methods:

```
public interface EncoderLogic
{
  public byte[] encode (byte [] content, String [] args) throws IOException;
  public byte[] decode (byte [] content, String [] args) throws IOException;
}
```

- `encode()` converts the data received from the server to a special format prior to its being sent to the client.
- `decode()` reverses the operation performed by the `encode()` function.

3) Using the Template API: Once the CL is ported to this interface, an encoding reAgent can be created with the following code:

```
EncoderTemplate reagent;

String   encoder     = "Compress.class";
String[] encoderArgs = null;

reagent = new EncoderTemplate(encoder, encoderArgs,
                              "middleman.ucsd.edu");
```

The constructor method of `EncoderTemplate` creates a reAgent that will instantiate an object of the CL class with `encoderArgs` as arguments. In this example, the CL does not take any arguments, so `encoderArgs` is null. It will also then launch the reAgent to the location "`middleman.ucsd.edu`", provided as a parameter. The reAgent is now at the remote host (the "reAgent host") and is ready to communicate with the client.

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

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

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

At this point, we have a working reAgent that compresses the server data before sending it to the client. The client-side logic of the template (pre-defined as part of the EncoderTemplate code) then decompresses the data and returns it to the browser. Neither browser nor server know that any compression occurred — the browser knows of the reAgent, but not its internal function, while the server believes the reAgent is the actual browser. In this way, the reAgent homogenizes the client environment for servers while requiring no effort and knowledge on the server's part.

B. Example 2: Customizing Communications

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

For the reAgent to use a custom protocol, the Templates package defines a `Protocol` interface that all custom protocols must implement:

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

After writing an implementation of the custom protocol with this interface, the protocol class file is passed to the Template constructor, as follows:

```
String cProtocol = "myClientProtocol.class";
String sProtocol = "myServerProtocol.class";

reagent = new EncoderTemplate(encoder, encoderArgs,
                              "middleman.ucsd.edu",
                              cProtocol, sProtocol);
```

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

C. Example 3: Integration into Traditional Applications

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

```
public void main () {
  Protocol HTTP;                          // implementation of HTTP

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

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

```
public void main () {

  EncoderTemplate reagent;

  String   encoder     = "Compress.class";                  *
  String[] encoderArgs = null;                              *

  String   cProtocol   = "HTTP.class";                      *
  String   sProtocol   = "HTTP.class";                      *

  reagent = new EncoderTemplate(encoder, encoderArgs,       *
                                "middleman.ucsd.edu",       *
                                cProtocol, sProtocol);      *

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

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

D. Example 4: Simple Data Encryption

Little work was needed to create the compressing reAgent beyond obtaining a suitable CL and interfacing it with the appropriate behavior template. However, one could argue that one could simply build a data-compression agent and use it for any client in a similar situation. But not all users face the same scenario as the user in Example 1. Take another laptop user, this time one who is unconcerned about low bandwidth,

but is concerned about the privacy of transactions carried out over the wireless link, particularly with a server that does not support a protocol secure enough to guarantee privacy. Here, the user would find a reAgent useful, but for a different reason. The reAgent can intercept the server data before it reaches the insecure wireless link, encrypt the data for privacy, send the encrypted data to the client, and have the client decrypt the transmission once the link is passed.

Note that this is the same type of behavior as in the first example, except that instead of compressing/decompressing, it is encrypting/decrypting. In each case, a reAgent receives data from the server, changes it to compensate for a deficiency in the client connection to the network, sends it to the client, and then undoes the change. With behavior templates, the shared parts of the behavior are already written. The programmer only needs to provide a suitable CL to customize the reAgent's behavior towards the needs of the client (in this case, privacy). For example, here the CL is chosen to be RSA [8]. An implementation of RSA is written in `RSA.class`, ported to the `EncoderLogic` interface, and sent to the template as a parameter to create a reAgent that behaves in a predictable, useful fashion:

```
EncoderTemplate reagent;

String   encoder     = "RSA.class";
String[] encoderArgs = { new String(publicKey) };

reagent = new EncoderTemplate(encoder, encoderArgs, "middleman.ucsd.edu");
```

Note that the only changes are in the class file for the encoder and its arguments (the public key used by the algorithm).

III. Characteristic Behaviors

We now present the core set of characteristic behaviors, with their defining characteristics. Each behavior is codified by a *behavior template*, which defines an interface for implementing a reAgent specialized by customizing logic. As described in Section II, a programmer that wishes to build a reAgent must first identify the general type of remote behavior it is to exhibit, select the corresponding behavior template, and incorporate the CL (application-specific Customizing Logic). The result is a reAgent that exhibits the general behavior in an application-specific fashion. By using behavior templates, programmers gain a structured, easy-to-use approach to building reAgents that lends itself to reuse over a wide variety of clients. The programmer is only responsible for obtaining the CL and incorporating the code into a reAgent via the template interface.

A. Filter

The Filter behavior (Fig. 3) is used whenever there is a need to reduce data sent from the server to the client by extracting and discarding some portion. The CL is the application-specific algorithm that defines *how* to reduce the data, i.e., what to extract and discard.

The Filter behavior is designed for scenarios where some of the data is extraneous, unimportant, or unusable. This scenario is highly relevant for clients that have limited capabilities, such as small battery-powered wireless devices (e.g., PDAs). General features include limited network bandwidth as well as low-fidelity rendering of data, so filtering the data before sending it to the client would conserve bandwidth without significantly impacting the perceived quality of the data.

1) Logic Interface:

```
public interface FilterLogic
{
  public byte[] filter (byte[] content, String [] args)
    throws IOException;
  public boolean isFilterable (Response responseStruct);
}
```

- `filter()` filters the server data in a client-specific fashion.
- `isFilterable()` tests to see if the response is filterable by this custom logic. (For example, an image filter should not be used on text.)

2) Logic Outline: The logic outline is the overview of what happens when the `process()` method is called. Each logic outline is split into client-side and remote-side components. The client-side component runs on the client; the reAgent-side component runs on the reAgent host. Thus, while this code is all encapsulated as part of the reAgent, part of it actually runs at the client while the other part runs at the "remote location" of the reAgent. These complexities, along with all the details of low-level communication and synchronization of processing, are all abstracted away from the programmer. However, to provide more control over communication, the programmer is able to install a client-specific protocol that the client-side component uses to communicate with the reAgent-side component, and a server-specific protocol used by the reAgent-side component to communicate with the server. These protocols, as explained in Section II-B, are defined by the template constructor function.

In the Filter, the logic outline is simple: the reAgent tests to see if the Filter algorithm should be applied to the server data, runs the Filter, and sends the filtered result back to the client.

```
client-side logic                reAgent-side logic
=================                ======================

deploy (reAgent)

cProtocol.send (query)           query = cProtocol.receive()

                                 ----------------------------------
                                 sProtocol.send (query)
                                 resp = sProtocol.receive()
                                 if (cLogic.isFilterable (resp))
                                   filteredResp =
                                     cLogic.filter (resp.content, args)
                                 ----------------------------------

response = cProtocol.receive()   cProtocol.send (filteredResp)

return response
```

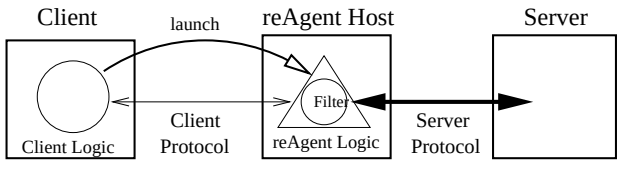

Fig. 3. The Filter Behavior

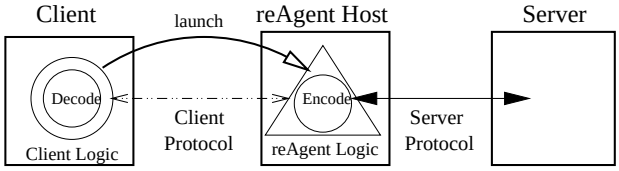

Fig. 4. The Encoder Behavior

B. Encoder

The Encoder behavior (Fig. 4) is used whenever there is an advantage derived from transforming server data at some intermediate point, sending it to the client, and then restoring it to its original format. Consider, for example, wireless clients with security concerns over their connecting wireless link to the Internet. Radio-wave transmissions for wireless communications are susceptible to eavesdropping, so security-sensitive applications running on wireless clients would use this behavior to encrypt their transmissions at an intermediate site before sending it over the insecure portion of the network. Another common, useful application that uses this behavior is compression of server data for transmission over an atypically low-bandwidth network segment. After passing this segment, the data is uncompressed to its original format. Thus, for the Encoder behavior, the amount of client-side processing is significantly more than what occurs in the Filter behavior.

1) Logic Interface: The logic interface for the Encoder was described earlier in Section II-A.2.

2) Logic Outline: The reAgent encodes the server response at the intermediary, sends the encoded data to the client, and then decodes the encoded data at the client.

```
client-side logic                   reAgent-side logic
=================                   ==================

deploy (reAgent)

cProtocol.send (query)              query = cProtocol.receive()

                                    ---------------------------------
                                    sProtocol.send (query)
                                    response = sProtocol.receive()
                                    encResp = cLogic.encode (response)
                                    ---------------------------------

encResp = cProtocol.receive()       cProtocol.send (encResp)

-------------------------------------
response = cLogic.decode (encodedRes)
-------------------------------------

return response
```

C. Monitor

The Monitor behavior (Fig. 5) is designed for use in applications that have a need to frequently examine the state of a remote object (on a far-away server), and trigger an action when some special state is observed. This action will often involve some measure of communication with the server. The combination of specific trigger action and monitoring algorithm form the CL. The CL is also responsible for triggering the sending of results to the client.

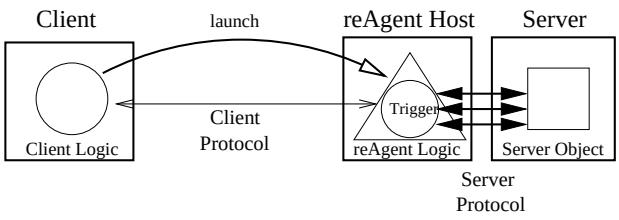

Fig. 5. The Monitor Behavior

The Monitor behavior allows the monitoring process to be located at a site closer to the object that is being monitored, which will improve the interaction time for communicating the trigger action to the server. This is important for applications that require a quick response to sudden changes in the environment. Examples include stock trading or bidding in real-time online auctions.

1) Logic Interface:

```
public interface MonitorLogic
{
  public long calcNextQuery (Response responseStruct, long lastQueryTime);
  public boolean testResponse (String [] args, Response responseStruct);
  public byte[] generateTriggerAction (Response responseStruct);
}
```

- `calcNextQuery()` returns the next time the monitor should make another query.
- `testResponse()` tests to see if the server response has produced a trigger state.
- `generateTriggerAction()` generates a trigger action to be sent to the server once the trigger state has been reached.

2) Logic Outline: The reAgent 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, it executes the trigger action, and returns a result to the client.

```
client-side logic                   reAgent-side logic
=================                   ==================

deploy (reAgent)

cProtocol.send (query)              query      = cProtocol.receive()
cProtocol.send (queryParam)         queryParam = cProtocol.receive()

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

                                        /* pause before checking */
                                        queryTime =
                                          cLogic.calcnextQuery (queryParam)

                                        sleep (queryTime - currentTime)

                                        /* check remote object */
                                        sProtocol.send (query)
                                        response = sProtocol.receive()

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

                                    /* perform one-time action */
                                    sProtocol.send (cLogic.generateTriggerAction())

                                    result = sProtocol.receive()
                                    -----------------------------------

result = cProtocol.receive()        cProtocol.send (result)

return result
```

D. Cacher

The Cacher behavior (Fig. 6) is used for storing recently retrieved server data at a nearby location with the expectation

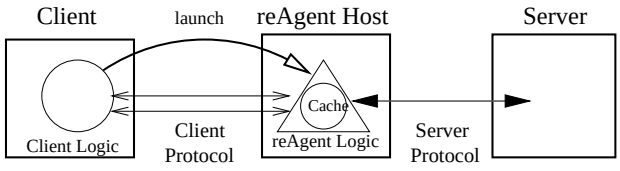

Fig. 6. The Cacher Behavior

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, such as Web browsing.

1) Logic Interface:

```
public interface CacherLogic
{
  public void create(String [] args);
  public String hash (byte[] request);
  public Response lookup (String key);
  public void replace (String key, Response responseStruct);
}
```

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

2) Logic Outline: In operation, the client launches a reAgent to the reAgent host to intercept requests from the client to the server and decide whether or not to pass along the request. If the request has not been made recently, the reAgent associates the request with a "key" (derived from the protocol) and passes the request through to the server. When the server responds to the reAgent, the reAgent associates the data in the response to the key of the request and stores both items in a database, i.e., the cache, before sending the response data back to the client. The data in the cache is kept for a maximum period of time called the "keep period." During the keep period, if the client makes a request that matches a key in the cache, the reAgent will bypass sending the request to the server and immediately return the associated cache data to the client.

The reAgent is in charge of inserting, removing, and retrieving data contained within the cache. Insertion of data happens whenever the server sends the reAgent a response. Data and its corresponding key are removed whenever the keep period for that piece of data expires, 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 within the cache. While the reAgent 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.

```
client-side logic                reAgent-side logic
=================                ================

deploy (reAgent)

cProtocol.send(request)          request = cProtocol.receive ()

                                 (pattern of behavior)
                                 -------------------------------
                                 key = cacherLogic.hash (request)

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

response = cProtocol.receive()   cProtocol.send(response)

return response
```

E. Collator

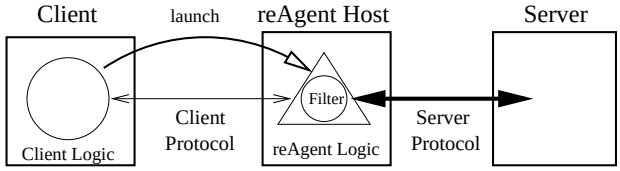

Fig. 7. The Collator Behavior

The Collator behavior (Fig. 7) is used in applications that need to transmit the same message to multiple servers, and then operate on the multiple results to return one result (collation). A typical application that exhibits this behavior is a comparison shopper that queries different sites with the same question and returns the "best" result. While most shopping agents focus on price, different users may have different ranking criteria, encapsulated in the CL.

1) Logic Interface:

```
public interface DistLogic
{
  public void wait (int replies);
  public Object collate (Response[] responses);
}
```

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

2) Logic Outline: 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 and collates the data in an application-specific way to the client. For example, the reAgent may only wait for the first response from any server, or some bounded number of responses, or even responses from all servers within a timeout period.

```
client-side logic                reAgent-side logic
=================                ================

deploy (reAgent)

cProtocol.send (query)           query      = cProtocol.receive()
cProtocol.send (serverList)      serverList = cProtocol.receive()
cProtocol.send (minReply)        minReply   = cProtocol.receive()
cProtocol.send (timeSlice)       timeSlice  = cProtocol.receive()

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

                                 for (s = 1 to n)
                                   sProtocol.connect (serverList[s])
                                   sProtocol.send (query)

                                 replies = 0

                                 for (i = 1 to n)
                                   spawn Thread that runs :
                                     resp[n] = sProtocol.receive()
                                               [blocking until timeout]
                                     replies = replies + 1
                                     [replies is a synchronized var]

                                 cLogic.wait (replies);

                                 data = cLogic.collate (resp)
                                 -----------------------

data = cProtocol.receive()       cProtocol.send (data)

return data
```

IV. Experiment

We have implemented our behavior templates in Java, making use of a locally-developed mobile code system called Java Active Extensions (JAE), a bare-bones implementation

of one-shot code mobility using Java. For further information on the JAE system, see [9]. To experimentally evaluate the overhead introduced by reAgents, we implemented the compression example described in Section II-A, based on the Encoder behavior template. We show that the overhead is low, especially when taken relative to the performance gains derived by a compressing reAgent.

A. Experiment: ZLIB Data Compression

In this experiment, the public-domain ZLIB compression algorithm was used to compress the server data received before sending it over a low-bandwidth connection to the client, where it was then decompressed.

1) Environment: In the following experiment, the following conditions applied:

- The client was a desktop PC PII–300.
- 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 low-bandwidth connection with effective bandwidth measured at 10–15 KB/s (KB = kilobytes). The reAgent host and data server were on the same subnet, so there was little overhead from network latency (thus allowing us to isolate observed overhead to our system). The regular bandwidth between the reAgent host and the server was measured at approximately 800 KB/s.

A fixed cost that needs to be paid at least once per reAgent creation is the launch overhead (the time it takes for the reAgent to be launched from the client to the reAgent host). The mean launch overhead in JAE 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, compression), and sending to the client (which then does the decompression).

2) Setup: 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 requests for actual PDF files (technical papers), 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 compression ratios, rather than canned ones that might be biased in favor of certain client-specific algorithms.

3) Results: The results, compared to a non-compressing Web browser, are summarized in Fig. 8.

For most of the files, the encoder exhibited good compression ratios, reducing end-to-end times by 30–75%. (The variable performance gain was dependent on how effective the file compression was.) The exception was the 10KB file,

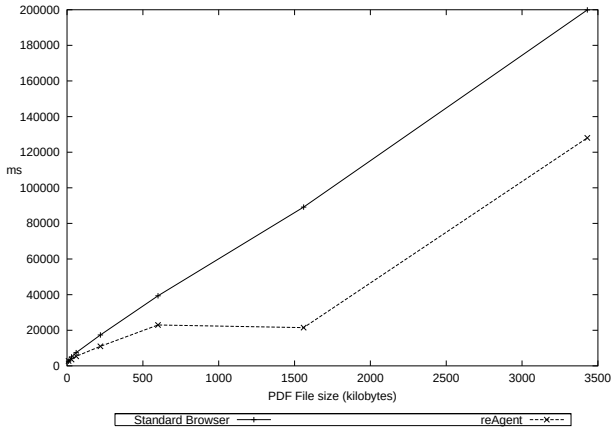

Fig. 8. End-to-end comparison times

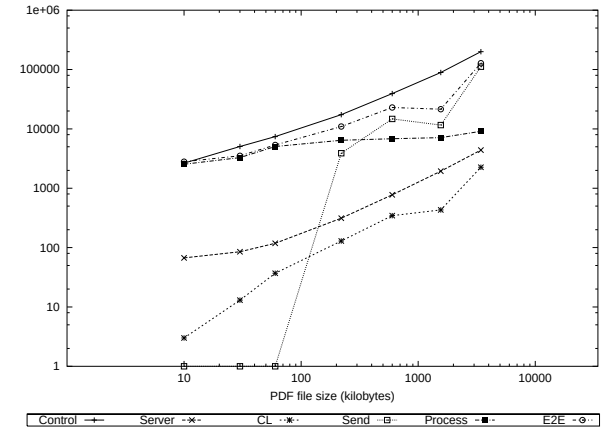

Fig. 9. Overhead from encoding a range of files (log-based)

where the gain from compressing the data sent over the limited bandwidth link did not compensate for the reAgent processing overhead. However, the encoder provided superior performance to the non-compressing client/server approach for files greater than 10KB. An obvious optimization would be for the reAgent to not compress small files, as the benefit does not outweigh the cost.

We timed different parts of the reAgent while encoding 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. 9, 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 encoding/decoding, processing, or moving the data from the server to the reAgent host were all minor and scaled well.

In this figure,

- **Control** represents the end-to-end processing time for a standard (i.e., non-reAgent) client/server implementation.
- **Server** represents the time it took for the reAgent host to download the file from the server (over a typical high-bandwidth connection).
- **CL** represents the time for the CL to compress.

- **Send** represents the time it took for the client to download the file from the reAgent host.
- **Process** represents the processing time of the reAgent.
- **E2E** represents 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 Encoder behavior template not only imposes little overhead, but can provide significant improvements in performance over a traditional client/server application.

V. Related Work

The customization of applications for improved performance has seen a variety of past research 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 and explain how our approach differs.

A. Active Networks

The active network area of research, as described in [1], argues for putting customizing logic at the network level with the use of programmable packets, or "capsules", that can change the behavior of the network. Active networks are able to effectively support a wide variety of client devices, but, because they are based in the network, operate at the expense of other applications that do not need them. In this paper, we chose a client-based approach because it does not impact the rest of the network as severely. Some active networks have been implemented by researchers, the most prominent being ANTS [11], NetScript [12], and SwitchWare [13]. The ALAN project [14], which moves the active network functionality into the application-level, addresses the issue of network deployment, but does not provide a general, structured method for building applications, the subject of this paper.

B. Dynamic Proxies

Another customization solution lies in the use of proxies, which act as intermediaries between client and server. Proxies are different from our approach in that they are not necessarily mobile (movable from site to site), and thus tend to be part of the existing infrastructure rather than originating from the client in response to a certain problem. While traditional proxy applications concentrate on caching Web results for improved performance and for controlling Internet access through firewalls [15], there has been some work done on proxies that actively customize application behavior (dynamic proxies). In [16], the idea of an Active Cache, or a dynamic proxy that acts to help improve caching for dynamic Web objects, was first proposed. In Active Cache, content providers provide specialized code in the form of a cache applet that intermediate caching servers execute to produce a new version of the cached object. More recently, [17] 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, server-independent customization solution. The reason for this goal is that new clients with correspondingly different demands are continually being created, and it is a difficult, if not impossible, task to anticipate every potential variation in client capabilities. Even if such variations were able to be anticipated completely, not every server has the resources to handle every potential variation. And if the server providing a unique service does not accommodate the client, the client has no recourse in a server-based customization scenario; it is completely dependent on the server for a solution to its particular problems.

C. Client-based Customizers

In order to be less dependent on the server, customizers with more client-side support have been developed. Reference [18] describes the implementation of a client-proxy-server framework that supports the on-demand downloading of custom filters (the customizing logic) to a proxy. The proxy then executes the filter on communications from the server before passing it onto the client. Unlike our work, this framework focuses on filtering applications instead of all types of applications that could benefit from mobile code. A more flexible Web-oriented customization scheme is detailed in [19], 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. However, it is optimized for use over an HTTP client/server connection and not a more generic client/server connection. Finally, the Active Names project [20] 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. Our work focuses on avoiding server participation, and thus differs in that the client must introduce the customizing logic, with the ability to apply it to any type of network.

D. Mobile Agents

A significant area of past research in client-based customization has been based upon *mobile agents*. Mobile agents are pieces of customizing logic that have a persistent identity, moving around the network to multiple sites. The IBM Aglets Workbench [21] and the D'Agents project [22], from industry and academia respectively, are prominent examples of systems that support the execution of mobile agents. A fuller description of these and other important agent systems, as well as the current state of mobile agent research can be found in [23].

Mobile agents provide a robust solution for addressing the problems of client heterogeneity: they are both deployable and scalable. However, despite having several years for the idea to incubate, mobile agent-based applications are rare. This is not due to lack of theoretical value: [24], [25], and [26] describe applications which take advantage of mobile agents. But, value notwithstanding, few applications based on mobile agents are in widespread use. Most application programmers are either unaware of the paradigm of mobile agents, or uninterested in handling the details necessary to support client-specific desires. Thus, our work differs from previous mobile agent literature by concentrating on a method that reduces the complexity of building agent-based applications.

VI. Conclusion

In this paper, we described a means for developing remotely executing agents (reAgents) that allow Internet applications to be customized to derive performance (or other) benefits for heterogeneous clients that are resource-limited, such as wireless clients. The approach is to use behavior templates, which abstract away many of the complexities of mobile code systems. When a developer uses behavior templates to build reAgents, Internet applications are easier to build due to pre-coded support for the movement, communications, and general processing functions used by that application's general behavior.

Our main conclusions from this work are as follows:

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

Future avenues of research include identifying and implementing more core behaviors, obtaining performance numbers for other basic template implementations, and exploring the possibility of dynamically combining behavior templates to easily create applications with more complicated behaviors.

References

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

[2] WAP Forum, *Wireless application protocol 2.0 specification*, http://www.wapforum.org/what/technical.htm

[3] 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.

[4] J. W. Stamos and D. K. Gifford, *Remote Evaluation.* ACM Transactions on 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] P. Deutsch and J-L. Gailly, ZLIB Compressed Data Format Specification version 3", RFC 1950, Aladdin Enterprises, May 1996.

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

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

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

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

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

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

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

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

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

[17] 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.

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

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

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

[21] 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.

[22] 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.

[23] 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.

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

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

[26] 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.