WebVector: Agents with URLs*

T. Goddard, V.S. Sunderam
Department of Math and Computer Science, Emory University
{goddard,vss}@mathcs.emory.edu

Abstract

Emerging web applications demand considerable network resources and, especially in science and engineering, considerable computational resources. The use of agents can eliminate bottlenecks in both areas, provided that their methods of computation and communication are not overly restricted. WebVector is an agent system that combines a flexible communication model with URL-based agent identification. Within this framework, resources can be centralized to support the current web-browser security model or distributed for efficiency. Interfacing with existing software can be as simple as adding a hyperlink to a file. We concentrate on cooperative applications: uploadable active resources, flexible multicast for maintaining shared-space state, and a variety of applications from distributed computation.

1. Introduction

A group of chemists direct their web-browsers to a page posted to a high-bandwidth server. Besides descriptive information in the form of text and tables, the page contains a shared note-taking applet and an embedded virtual reality (VR) representation of a molecule. Each chemist is represented in the VR world by an "avatar" reflecting their position and view. The molecule is not a simple ball-and-stick model; depicted is the solvent accessible surface surrounding a protein that the chemists fold in real time. As more collaborators join the group, performance actually improves as each web browser loads an applet that contributes to the solvent accessible surface computation. Agents on the server maintain the shared notebook, the state of the VR world, and the allocation of computational tasks. In the meantime, other agents have traveled to protein databases, looking for correlations, possibly bringing new molecules into the scene depending on their discoveries.

Active documents, transparently distributed computation, and efficient multicast are just a few examples of the sorts of qualitative and quantitative benefits we would like to bring to users through agents. By discussing agents in general and our system in particular, we hope to demonstrate the strengths of our system as well as convey a few ideas that would be useful in general. We will begin by explaining briefly what our working definition of "agent" is and how we addressed the issues common to all agents in the design of our system. Then we will look at a few other agent systems and move on to a more detailed examination of the architecture of ours. That will prepare us for a few implementation details and a complete (but simple) example showing WebVector in use. We conclude with a few example applications and possible future developments.

An "agent" can briefly be described as "one who acts for another". Rather than interpret this in the domain of artificial intelligence [1], we define agents as follows. *Agents* are programs sent by an entity, or one of its agents, to a remote system. As such, they cannot exist alone, they require a network of "agent hosts"; these are the computers to which agents are transmitted and thereupon executed, possibly creating and transmitting agents to other agent hosts themselves, or possibly just communicating with their source to return results (Figure 1).

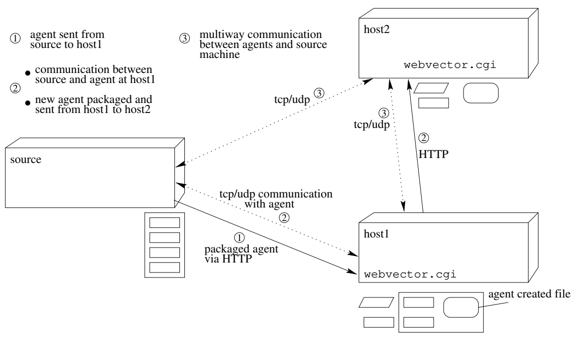

Figure 1. WebVector Agent life-cycle

Most researchers specialize this idea further to include *migration*; they insist that an agent be able to halt its execution, package and transmit itself somehow, then resume execution on a remote machine. This is appealing, but we

*Research supported by Army Research Office grant DAAH04-96-1-0083 U. S. Department of Energy Grant No. DE-FG05-91ER25105, and the National Science Foundation Award No. ASC-9527186.

do not see it as fundamental to every agent model. Our approach is to be less rigid; we do not wish to enforce specific methods of migration or communication as such methods may not be universally applicable. With our system, an agent can migrate by sending its executable and data capturing its "state" to the new location. The executable that arrives at the new location simply reads in the state data as its first operation.

Any particular implementation of agents must restrict migration, communication, and host access for reasons of security and practicality, the most pressing concern being security. Providing for the security of the machine hosting the agent is not a new problem; its solution is an integral part of the design of any time-sharing system. This problem is far from solved, but many current solutions are reasonable. A more novel set of problems is raised by the agent's abilities to migrate and communicate. These abilities bring the potential for the agent to attack systems that never had any intention of hosting agents. Moreover, the agent's potential for replication paves the way for attacks of unprecedented magnitude.

Since migration requires communication, it follows that the majority of security needs can be addressed by considering communication alone. Let us then consider the problem of protecting remote machines from agent communication (especially important when those "remote" machines are inside the same firewall as the host machine). Most agent systems address this indirectly by providing only a proprietary communication mechanism – existing software cannot be attacked by such agents because the proprietary mechanism cannot emulate existing protocols. Unfortunately, specialized protocols often limit the range of applications. For instance, agents that communicate through an offline batch communication mechanism would have a very difficult time manipulating an audio stream carried over UDP. Instead, our general security policy is to restrict communication to only those parties who desire it (note, however, that we allow agents unrestricted communication to web servers). This security policy provides protection for existing systems as well as agents themselves. The intent is to make being an agent host no more of a risk than using applets in a well-designed web browser.

2. Agent Systems Comparison

Agent systems can be distinguished by migration and communication mechanisms, security policies, and supported languages or environments. Most systems do not discuss security in detail, though, because their migration and communication mechanisms do not interface with existing software. Still, it is important to realize that an established agent system (where establishment means that agents are performing tasks of value) will develop its own security problems: hostile agents will appear and their intended targets will be only other agents, not the host systems.

Web browser Applets. Whether web browser applets should be considered agents or not is largely a matter of opinion, but if they are not agents, a strong case could be made that this is so largely because of the standard applet security policy (an applet may open a socket only to its machine of origin and may not listen for any connection). In any case, an applet could act as an agent host.

AgentTCL. AgentTCL [7] agents migrate via a modified TCL interpreter facilitated `agent_jump` and are provided with message based communication as well as a named stream between two agents through `agent_meet`. The dedicated AgentTCL server accepts and executes incoming agents, accepts and buffers incoming messages and connection requests, and enforces the security policies.

Infospheres. The infospheres project [3] has more encompassing goals than many others, "the Caltech Infospheres project develops theories, methods and tools to support infospheres" where an infosphere is "our current state and a set of interfaces through which we interact"; in more mundane terms, they are building and theorizing about a distributed object system. They currently use Java with communication through remote method invocation and synchronous/asynchronous message primitives.

TACOMA. TACOMA [8] is based on TCL and a single (but complete) communication mechanism `meet`. An agent communicates with another not by sending a message, but by traveling to the remote TACOMA host and exchanging information there, carrying its state and other data along in "folders". Agent hosting is mediated by a background firewall agent.

IBM Aglets. The aglet system [9] provides an environment for dispatching and hosting java agents. Aglets exchange messages by invoking the methods of "proxy" agents (similar to java RMI stubs) and are capable of halting themselves and resuming execution on a remote host. Filesystem access can be based on whether an aglet is "trusted" (originated on the host machine) or "untrusted" (originated elsewhere). Aglet transport is handled by a protocol superficially different from but incompatible with HTTP.

3. WebVector Architecture

We see agents as a natural part of the World Wide Web and have allowed this to guide the design of WebVector. For example, it seems reasonable to send an agent to a remote location, have it perform some computation suitable to that location, then as a result of that computation, produce a document (perhaps in HTML) that we retrieve upon completion. The document that we retrieve would be most naturally retrieved and viewed in a web browser, hence it makes sense for the agent's files to have URLs associated with them (as a matter of course). We take this idea a little further, as files are not the only resources served by URLs. For instance, the "telnet:" protocol identifies a TCP-port on a given machine. Thus, if an agent wishes to listen on a certain port, it can make the URL available for that connection. Moreover, the agent can identify the purpose of the port in the text of the hyperlink for the URL; for instance `<A HREF="x-tcp://host.edu:1234">agent status</A>`. Notice that the above protocol is given as "`x-tcp`" rather than "`telnet`". We propose two URL protocol strings: "`x-tcp`" and "`x-udp`" for such purposes.

This mechanism is a critical component of our agent context as it assists agents and other entities in finding each other (without the difficulties of arranging pre-defined port numbers) and provides protection for remote machines and other agents.

WebVector is composed of essentially three components: a cgi-bin script called `webvector.cgi` that unpacks and executes agents, a security policy enforced through our Java `SecurityManager`, and a collection of core routines and classes in a Java package called `WebVector`. The agent expects to find all of these components on the host system, although it will interact directly only with the package `WebVector`. As we fine-tune the code that makes up the WebVector package, we hope to move a little more from the host machine classes into the classes that the agent brings along with it. This is of benefit for two reasons: in terms of security, there is no question that a smaller host system is easier to analyze; in terms of maintainability, putting more code with the agents itself ensures that the agent always finds the version that it expects.

We provide a class called `NamedServerSocket` whose constructor takes the string `socketname`; this class interfaces with our SecurityManager and places an entry as described above (with hyperlink text from `socketname`) in the agent's `services.html` file. Besides providing a way of indicating how to communicate with an agent on the appropriate channel, we use the above mechanism to apply a security policy as follows. An agent is only allowed to open sockets corresponding to the URLs found in the hyperlinks of other agents' `services.html` files which it has discovered through `Tracker.lookup(URL agentURL)` calls. This provides two levels of protection. First, and most important, it provides protection for existing network services; for instance, only if the system administrator puts a hyperlink to port 79 in a `services.html` file in an agent's directory can another WebVector agent talk to the finger daemon on his machine (on the other hand, this illustrates how easy it is to allow WebVector agents to communicate with existing software when that is desired). Second, agents can obtain protection for themselves (at least from other agents) by placing hyperlinks to some of their services in `services.html` files in restricted or "invisible" directories. Such invisible directories cannot be seen by other agents (unless they know the name of the directory) thereby giving agents a level of security comparable to standard login passwords.

4. Authentication and Security

Authentication. WebVector accepts two categories of agents, `SIGNED` and `UNSIGNED`; these denote files or agent components that have been received and verified as digitally signed or received with no such authentication respectively. An agent is unlikely to be able to protect itself from an untrustworthy host, but there are two senses in which a signed agent can be useful. If the host is trusted, then it can be trusted to have verified that the `SIGNED` agent is indeed acting on behalf of the creator. We can then view the agent as an extension of the creator's machine. If the host is not trusted, and we are only interested in the files, then we can still retrieve the digital signature along with the files and check the signature ourselves. Note that an agent should not attempt to produce signed files at a remote location as an untrustworthy host could disassemble the agent, learn the private key, and pass its own files as being signed by the agent's authority.

Using a SIGNED agent has the added benefit of providing a reserved name space. UNSIGNED agent names are allocated on a "first come, first serve" basis with the first user of a particular name on a particular machine being rewarded by a randomly generated "cookie". This ensures that they are the only ones who can install further resources with that agent name (up to the security of their HTTP connection) on that machine, but other machines may have agents of the same name but of entirely different origin. Agents signed using PGP via the MIME multipart/signed content-type ([6]; [5]) are assigned URLs starting with `http://hostname/webvector/SIGNED/PGP/`*keyID*/ where *keyID* is the keyID of the PGP digital signature (expressed as eight hexadecimal digits). Since it is highly improbable for two keys to have the same keyID, this effectively reserves a portion of the agent name-space on all machines. WebVector will attempt to verify the keyID and signature using its own key ring or through consultation with a collection of key servers (both set up by the system administrator) and will not install the agent unless the signature is valid. Observe that we have not solved the problems related to maintaining vast numbers of public keys, but there are uses for digital signatures nonetheless.

Host Security. One of the great advantages of agents is that the computation can be placed near the data, but this advantage can be fully realized only if agents are able to read and write files. WebVector agents are allowed to read and write files in the same directory as that pointed to by their base URL. They may read also files outside their own directory, but only indirectly via the web server (like any other entity on the network) thereby providing privacy for agents on the same machine.

The restrictions on URLs for agents must be somewhat strict. They must begin with `http://hostname/webvector/` and this must be the case as `services.html` files under the webvector base URL are taken to speak for the allowed communication access for the whole machine. User-level WebVector installations could expose fragile communication resources on machines where the administrator had no knowledge that it was being opened to attack by remote agents.

Network Security. As was described in the section on architecture, WebVector agents cannot open a socket unless the destination machine has the desired port registered in the `services.html` file of some agent and the agent opening the connection has performed a `Tracker.lookup` on the agent using that port. This protects existing software and provides a mechanism through which agents can protect themselves.

Agent Security. By using "invisible" directories and files (most web servers provide a facility for returning a particular document only when its URL is given explicitly) agents can keep a selection of their resources secret from the outside world. Unfortunately, other users of the host machine may be able to read the file system directly, thereby bypassing the web server's efforts. For this reason we suggest that the web server be configured in such a way that only the system administrator and web server itself have read or write permission on agent files – the fact that web-accessible files are world readable through the server does not imply that those files need be world readable through the file system. Since agents are prohibited from reading files outside their own directories by the WebVector SecurityManager, such a policy will provide fairly complete privacy (but still with the risks associated with any re-use of passwords).

Future Considerations. Further development of security policies is warranted. For instance, the massive parallelism that can be attained by agents potentially makes them dangerous instigators of "denial of service" attacks. Some agents might replicate uncontrollably simply due to programmer error. Perhaps agents could declare their communication and migration topology, giving each machine involved a chance to opt out if it would be subject to an inappropriate load. Network access outside the group of agent hosts could also be limited and such limits could be applied to the original agent and its children as a whole.

5. Implementation

Executables. Current agent systems seem to be focused around two languages: java and TCL. Both languages are highly portable and expressive, but we have selected java as the first language supported by our system because of security and performance. During our examination of security, we were impressed with the close integration of the java `SecurityManager` with the entire java system. Java may not be suited to a formal verification of security [4], but considerable resources are being applied to the problem of making a secure implementation of java, so a workable solution is likely. Performance is not critical for all agent applications, but better performance expands the range of media types usable by agents and brings the potential for network applications. In any case, it is desirable to use the host machine efficiently. Java is not currently thought of as a performance leader, but emerging "just in time" compilers should prove more than adequate for agent applications. We do not see the same performance potential in TCL.

Three types of java executables are currently supported: `Thread` (or just plain `Runnable` started up in a `Thread`), `Applet` (although calls to display routines may not have the desired effect as an agent does not have a display), and cgi-bin agents (expected to implement the `Runnable` interface). Agents expecting to function as cgi-bin agents must reside in a cgi-bin directory of their installation.

Packaging. Rather than invent a new data format for agents, we have chosen to interpret existing MIME [2] standards in an agent context. Agents may be composed of multiple components; this is served by MIME messages with the header `Content-type: multipart/mixed` [11]. Given the components of an agent, the host sytem needs to know which are to be executed; this is served by the header `Content-disposition: inline`. Also in the `Content-disposition:` header is a place for specifying a filename; this is naturally used for the filename of the particular component on the host system (subject to security constraints, of course). Once a component has a filename, it automatically inherits a URL. It is our expectation that the MIME standard will continue to increase in popularity and this will result in the production of a variety of tools that facilitate working with agents, even though those tools were not designed with agents specifically in mind. Further, we imagine future agents traveling by a variety of means – moving from one web-server to another, arriving in electronic mail messages, fanning out by multicast, etc. Anywhere that the MIME standard was adopted would become a potential pathway for agents.

Currently, the unpackaging routine (`webvector.cgi`) is a combination of ksh and perl scripts. We also have simple packaging utilities written in perl.

Transport. Once encapsulated in a multipart MIME message, an agent may travel by a variety of means, but our intent of providing agents with URLs makes one means clear: HTTP. The cgi-bin agent hosting program `webvector.cgi` receives an agent via a `POST` to the (example) URL `http://host.edu/cgi-bin/`

`webvector.cgi/UNSIGNED/`*agent-name*. The only point of interest is that we use component of the path following `webvector.cgi` to specify the agent's authentication as `SIGNED` or `UNSIGNED` and to specify its name.

Currently we use a simple perl script to initiate the transport of agents, but it would be straightforward to modify `webvector.cgi` to accept MIME multipart messages in the slightly less expressive format produced by web browsers (web browsers do not generally allow the user to set the `Content-disposition` header).

6. Usage

Let us illustrate what is involved in producing and activating a simple WebVector cgi-bin agent.

The following agent, `registerMe.java`, appends the CGI "query-string" to a file with the intended application being to gather a list of participants. It's true that `registerMe.java` looks like a typical cgi-bin program rather than something conforming to a specialized agent API; this reflects our goal that WebVector agents use existing code as much as possible.

```
import java.io.*;
import java.net.*;

public class registerMe implements Runnable {
  public void run () {
    String qs = System.getProperty("http.query_string");
    System.out.println("Content-type: text/html");
    System.out.println("");
    try {
      String line;
      RandomAccessFile f = new
          RandomAccessFile("../list.html","rw");
      f.seek(f.length());
      f.writeBytes(qs + "<BR>\\n");
      DataInputStream list = new DataInputStream(
          new FileInputStream("../list.html"));
      while ((line = list.readLine()) != null)
        System.out.println(line);
    }
    catch (IOException e) { System.out.println(e); }
  }
}
```

The HTML file `list.html` starts out as a very simple (one line) file: `<H1 ALIGN=CENTER> Current list of Participants </H1>` As participant names are appended to it, we produce a list of names below a centered title.

Compilation of `registerMe.java`. would produce `registerMe.class`. Both `registerMe.class` and `list.html` are packaged into a multipart MIME file:

```
Content-type: multipart/mixed;boundary=0.57776819635182619

--0.57776819635182619
Content-disposition: inline; filename="cgi-bin/registerMe.class"
Content-type: application/java

          (Java bytecodes go here)

--0.57776819635182619
Content-disposition: attachment; filename="list.html"
Content-type: text/html

<H1 ALIGN=CENTER>Current list of Participants</H1>

--0.57776819635182619--
```

This multipart MIME file is POSTed to the URL `http://host/cgi-bin/webvector.cgi/UNSIGNED/register` Then, the URL `http://host/cgi-bin/ webvector.cgi/UNSIGNED/register/cgi-bin/registerMe.class?Gauss` would add "Gauss" and return the current list.

7. Applications

Below are a few of our favorite potential applications for agents. Many of these applications are not realizable in other agent systems because of the system's use of a proprietary message-passing scheme. This has security and programming efficiency advantages, but we believe those advantages are outweighed in many cases by the flexibility offered by the more familiar sockets.

Uploadable Active Resources. The web is a tremendous publishing tool. One common situation is that of an author creating a document, then "uploading" that document to a web-server with both the bandwidth and visibility desired by the author. Unfortunately, this method does not lend itself well to publishing all types of resources. For instance, what if the author wanted to publish a database? Using our agent system, the author would upload the database, a cgi-bin agent, and an HTML page containing a form for communicating with the cgi-bin agent.

As another example, consider where the author wishes the viewers of the document to contribute to the state of the document, as in our opening example, where the viewers' avatars are part of a shared world. The author would upload data files for generating the world together with agents and applets for maintaining the shared world-state.

Flexible Multicast. Multicast is very useful in collaborative applications for maintaining a shared "state". The general means of arranging multicast is through the MBONE, but not every machine participates in the MBONE, nor is the MBONE algorithm always best suited for every multicast application. By uploading a network of agents, an author can create a multicast tree with nodes on machines of his choosing. Existing applications can be added into the tree (by a super-user) by entering their ports as URLs in `services.html` files.

Search Engines. Search engines are an integral part of daily life on the web, but currently must resort to (essentially) downloading entire sites. By sending an agent, producing a digest at the site, then later retrieving the digest, network usage could be greatly reduced. This is not a novel application of agents, but searching for resources is so commonly performed and so commonly cited as an application for agents that we would have been remiss without it.

Distributed Computation. Our target applications are scientific applications including visualization; therefore, efficient computation (especially from the user's perspective and not necessarily in terms of overall resource usage) is of great interest to us.

Agents can be of computational benefit in two ways:

one, by occupying many machines and thereby achieving parallelism; two, by residing near a remote machine of significant computational power (and perhaps not performing any computations itself). Let us look at the second case in more detail as the first case is straightforward.

Suppose an institution has a machine of significant computational power that they wish to make available on a limited basis. For security reasons, they do not want arbitrary programs from the outside world to run on this machine, but they are willing to allow a subset of LAPACK to be used. A natural thing to do would be to set up a cgi-bin script that accepted LAPACK arguments and returned the results. Unfortunately, many linear-algebra applications require recursion and this would entail an excessive number of trips to and from the web-server. By placing an agent host on the LAN with the LAPACK machine, an agent can act as a "controller" still using the cgi-bin interface, but paying only a LAN-worth of communication cost. Notice that the LAPACK machine itself is not expected to host agents (there is no reason to assume that java will run on a special-purpose architecture and certainly no reason why java would be efficient on such an architecture).

8. Discussion and Future Developments

We have tested WebVector with some simple computational, cgi-bin, and multicast relay agents with our next application goal being the molecular visualization system presented in the introduction. Our current implementation provides sufficient functionality, but many issues still remain. For instance, we do not have an expiry mechanism for agents. Ultimately, the decision of when to delete an agent rests with the host, but it is useful to provide the host with advice so that it can use its resources more efficiently. Further developments can also be expected as java itself evolves. For instance, the recent introduction of Remote Method Invocation and Object Serialization, besides allowing agents to be dealt with as a collection of distributed objects, greatly simplify the implementation of agent migration. In java's future, advances in Java Virtual Machine (JVM) schedulers will allow the host finer control over agent cpu-usage. A flexible agent system such as WebVector can benefit from many emerging technologies.

Security has been a consideration in every step of the design of WebVector, but one must keep in mind that an HTTP connection itself poses a rather weak link. Technologies such as Netscape's Secure Sockets Layer [10] appear to be a step in the right direction, but we wish to analyze the protocol further before recommending it. In any case, privacy and authentication are needed for HTTP connections in general. This provides a strong motivation for the community of web developers, so a workable solution is likely to appear. This solution will, in turn, bolster the security of WebVector.

9. Conclusion

The new software model that agents provide can be studied on a purely theoretical level with any of the agent systems considered, but we feel that WebVector is currently the best suited for actual applications for several reasons. Consider that the quantifiable benefits of agents are almost entirely due to the communication topologies and optimizations that they offer; WebVector provides (within security limitations) the same communication options as are available to any java program. By giving agents and their services URLs, we encourage hyperlinked indexes and directories that are simultaneously machine and human readable. Sometimes it is useful for an agent to lie dormant (using no resources other than disk space), be "awakened" with data for it to process (assisted by services present near the agent), then return the result and lie dormant again. This is precisely the application of cgi-bin agents, and such agents are very easy to implement within WebVector. It is important to study agents in abstract terms, but there are variety of applications that can make good use of agents now; WebVector was designed for exactly this purpose.

References

[1] Intelligent agents. *Communications of the ACM*, 7, July 1994.

[2] N. Borenstein and N. Freed. MIME (Multipurpose Internet Mail Extensions) part one: Mechanisms for specifying and describing the format of internet message bodies. *RFC1521*, 1993.

[3] K. Chandy, A. Rifkin, P. Sivilotti, J. Mandelson, M. Richardson, W. Tanaka, and L. Weisman. A world-wide distributed system using java and the internet. *submitted to IEEE*, 1996.

[4] D. Dean, E. Felten, and D. Wallach. Java security: From hotjava to netscape and beyond. *to appear 1996 IEEE Symposium on Security and Privacy, Oakland CA*, May 6-8 1996.

[5] M. Elkins. MIME security with Pretty Good Privacy (PGP). *RFC2015*, October 1996.

[6] J. Galvin, S. Murphy, S. Crocker, , and N. Freed. Security multiparts for MIME: Multipart/signed and multipart/encrypted. *RFC1847*, 1995.

[7] R. S. Gray. AgentTCL: A flexible and secure mobile-agent system. *Proceedings of the Fourth Annual Tcl/Tk Workshop (TCL 96)*, July 1996.

[8] D. Johansen, R. van Renesse, and F. Schneider. Supporting broad internet access to TACOMA. *Proceedings of the Seventh ACM SIGOPS European Workshop, Connemara, Ireland*, pages 55–58, September 9-11 1996.

[9] D. B. Lange and D. T. Chang. IBM aglets workbench programming mobile agents in java a white paper. 1996.

[10] Netscape. SSL. 1996.

[11] R. Toost and S. Dorner. Communicating presentation information in internet messages: The content-disposition header. *RFC1806*, 1995.