Integrating Software Agents into the HTTP Caching Infrastructure

Jesse Greenwald, Daniel Andresen
Kansas State University
Department of Computing and Information Sciences
234 Nichols Hall, Manhattan, KS 66506
785-532-6350

{jrg1455, dan}@cis.ksu.edu

ABSTRACT

Mobile software agents are an increasingly important programming model within the World Wide Web (WWW). Typically programmed in Java or another machine- independent language, the code and associated data have the ability to move their execution from machine to machine, often revisiting the same machine multiple times. In such a case, caching the agent's binaries can lead to a significant improvement in performance. However, existing agent systems generally do not address the issue of cache control via W3C standards. In this paper, we show the need for such systems and demonstrate this through combining IBM's Aglets agent system with the Squid proxy cache. We give experimental results indicating that a loosely coupled system adds only approximately 25 ms. to agent invocation in cache control overhead, while giving speedups of almost 700 percent in certain network configurations through the use of the WWW caching infrastructure.

Keywords

agents caching http squid aglets Java

1. INTRODUCTION

Mobile Intelligent Agents are autonomous units of logic that move themselves at will from host to host and communicate with the local resources and agents. One task agents perform well is parallel programming, where an agent is started and clones itself onto other free machines for processing. When the clones are done with their task, they move back to their parent and return their data. Agents can also reduce wasted network bandwidth [2]. Typical network interactions involve a user's querying a remote system for information, processing that information, and then querying the system again. This loop continues until the user is satisfied. Much of the bandwidth used is wasted on information that is irrelevant to the user. An intelligent agent, on the other hand, moves itself out to the remote system, executes the queries, and then moves itself back home with its results. This introduces a new performance issue: How much bandwidth is being used to move the agent around? If the agent is large, then it will exhibit worse performance with small queries due to the extra overhead. Potentially, more bandwidth will be used for moving the agent than for queries. Only when the query is big enough, and the agent small enough, will an agent be more efficient. To counter this problem, a caching proxy can easily be put in front of the agent system. Now, only the state of the agent will have to be transferred out on the Internet; the actual code for the agent can be found on the local cache.

2. ARCHITECTURE

Some agent systems, such as IBM's *Aglets*, have their own caching mechanism for code [1]. Unfortunately, they may not be designed to take advantage of W3C cache coherency protocols. When code is put into a system's cache, it may never be refreshed. Only when the cache is manually cleared will new code be downloaded. This approach drastically reduces the system's flexibility. We have been studying the effectiveness of using the coherency controls of HTTP in an agent system's cache. Specifically, we are using the Aglet agent system provided by IBM in conjunction with the Squid open-source caching project to explore the implementation of a system combining intelligent agents with a standards-compliant hierarchical caching system [1, 3].

We added an attribute to the cache entries containing the date the object was last modified. When an object is fetched via HTTP, the server returns the date that its object was last modified in a header directive (specifically the <Last Modified> directive) in the response. That date is then stored with the object in the cache and saved for further use.

We changed Aglets to allow for standards-compliant updating of its cache. When a class needs to be loaded, the cache is checked to see if it is present, but the server that holds the code will still be checked to see if it needs a new version. Using the HTTP header directive <If-Modified-Since>, Aglets will request that server return code only if the code has been updated. Opening a connection to the server every time significantly affects performance, even if no class is downloaded. To help compensate, a Squid cache is placed on the same machine as the Aglets server. When Aglets needs to check with a code server for new code, it will be processed through the Squid caching proxy. Because most of the caching responsibilities are offloaded onto Squid, only minimal changes are necessary to Aglets for full cache coherency support.

3. EXPERIMENTAL RESULTS

In our experiments, we tested to verify that the changes we made to Aglets actually altered the way that agents are loaded. We also measured the performance impact with respect to latency and processing times. All test results

are averaged over 100 runs.

The first test determined whether a change to an aglet's code would be noticed by the software. To perform this test, we started up one aglet environment. An aglet was put on a remote HTTP server. We created an instance of the aglet to ensure the HTTP server was set up properly. Without shutting down Aglets, we modified the code on the server and instantiated a new instance of the aglet. The newly created aglet was an instance of the modified code, which verified that our changes were effective.

We created four aglets that would ping themselves between two aglet environments a limited number of times and track the time it took to complete the task. We made the aglets different sizes to see how the server responded to varying loads. The experimental configuration consisted of a Fast Ethernet LAN connected to the Internet via cable modem, and an HTTP server residing on the Internet.

The second set of tests (Figure 1) measured the latency of Aglets under certain configurations. We compared the original version of Aglets with and without caching against the modified version with and without caching. The results show that the modified version of Aglets that included cache coherency ran at a nearly equivalent speed to the original version without caching. With agents that were 16KB, the original version ran at most 7.5 percent (11.19 ms.) faster than our modified system. Unless time is a critical factor, the difference in performance should not be an issue. When tested using an agent that was 64KB, the modified Aglets actually performed slightly faster (1.91 ms.). We also removed class caching from the original version of Aglets to see how the Aglets environment would perform without caching. When only a 1KB aglet is moving around, it takes 386.67 ms. for each hop. The added time is taken by opening a connection to the class server, downloading the class, and creating an instantiation of the class object. The performance drops significantly for a 64 KB aglet; requiring 1034.95 ms. for each move. The test was performed using broadband access, so downloading 64KB took a small amount of that time. Parsing the large class file and creating a class object for it took the majority of the time. Under this configuration there was no caching for the classes, which explains the time values. Our version of Aglets without a web cache outperformed Aglets with its cache disabled, indicating Squid is not the sole reason for our performance advantage.

We also compared the use of a Squid-based system to the standard Aglets configuration, which cannot take advantage of a cache hierarchy, but must download any necessary code from the original server. Using a 9.6kbs analog modem and a 64KB aglet, with a current version of the aglet on a local machine and the original host behind the modem connection, the time required for the agent to move between the two local systems was nearly seven times larger (83 s. vs. 12 s.) for the original Aglets system. Not surprisingly, utilizing the WWW cache hierarchy considerably improves the performance.

The primary contributors to system overhead are establishing a connection to the class server and fetching the results. Connecting to Squid and fetching the results takes on average between 25 and 30 ms. (about 17 percent) as shown in Table 1. The time Squid requires to process the

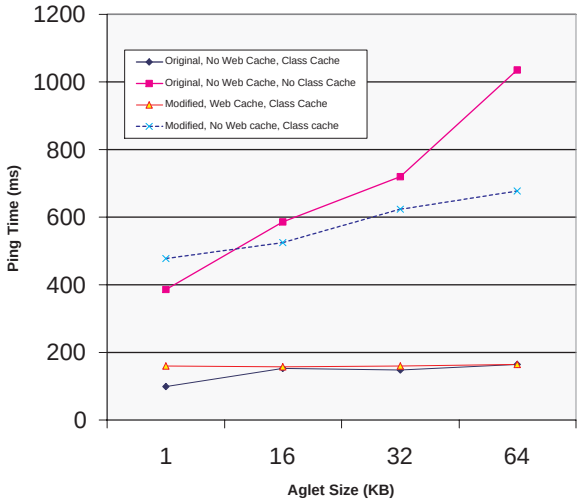

Figure 1: Performance comparison with and without caching.

query is minimal; on average it takes no longer than 0.5 ms. Memory and CPU usage for Squid are also minimal, < 1 percent of CPU and memory usage.

Size	Connect	Squid	Get Results	Total
1KB	10.81	0.55	16.52	27.88
16KB	12.34	0.525	13.0	25.87
32KB	11.81	0.445	12.1	24.35
64KB	10.61	0.505	16.53	27.64

Table 1: Overhead of Cache Coherency (ms)

4. CONCLUSIONS

We have presented a W3C standards-compliant system for caching code for mobile intelligent agents. Our modifications made it possible to update the internal cache of Aglets with new code, allowing up-to-date agents to be created without shutting down the system. Furthermore, pre-existing agents can have their code updated when they move to a different Aglets server. The new functionality makes Aglets more desirable for real-world implementations. The performance gap between our version of Aglets and the original is small, a minor penalty for significantly enhanced flexibility and access to the Web caching hierarchy. The original never performs more than 7.5 percent better with Aglets that are 16KB or larger, and ours performs better when using a 64KB aglet. When slow Internet connections are used, our configuration can significantly outperform (by over 700 percent in our case) the original version.

5. REFERENCES

[1] Lange, D., and Oshima, M., Programming and Deploying Mobile Agents with Java Aglets, Addison-Wesley, September, 1998. http://www.trl.ibm.co.jp/aglets/

[2] Harrison, C., Chess, D., and Kershenbaum, A., "Mobile Agents: Are they a good idea?", March, 1995. http://www.research.ibm.com/massive/mobag.ps

[3] The Squid web site, November, 2000. http://www.squid-cache.org