Aglets: a good idea for Spidering ?

Nick Craswell
Jason Haines
Brendan Humphreys
Chris Johnson
Paul Thistlewaite
[ANU]

---

1. Introduction

Many individuals and businesses now rely on the Web for promulgating and finding information, and in particular, rely on centralised search databases. The extent to which these databases reflect the "contents" of the Web in an accurate and timely manner is now under considerable doubt, and in any event, it is apparent that the methods used by the search engines for finding new and modified Web documents are not scaling well. To ameliorate these problems, we have been exploring the use of a "data push" model for notifying Web changes, to replace the current "data pull" model, which uses aglets (*aka* servlets or peerlets) to distribute the indexing task.

Aglets are objects with a thread of control, that can migrate autonomously between processors in a distributed environment. As currently proposed and implemented, aglets have very few of the properties of Persistence. But as they inhabit a similar conceptual space, their properties and applications present some interest to the persistence community. Aglets have unique identities, locally persistent data and methods (in the sense that the aglet can be deactivated onto disk and reactivated), and self-migration in a distributed environment, but they do not facilitate a universal name space. They appear to be modelled on a belief that the processors in the network, while needing to be sufficiently transparent to allow some cooperative processing, will remain sufficiently opaque to thwart a realisation of the Persistence Ideals.

2. Web Spidering

There are three ways of finding relevant information on the World Wide Web. One is to browse through document hyperlinks, manually or using an electronic agent. Another is to use a manually categorized URL list (a web directory). Finally, the most important method for information location on the web is indexing.

A central web index such as AltaVista or HotBot must gather information from the web and then make it available to enquirers. A major component is a *web spider*, which is responsible for finding and downloading current copies of the web documents to be indexed. Both the degree of coverage of an index and how up-to-date it is are determined by the architecture and efficiency of the spidering software.

2.1 Coming to Know of Changes

One way of viewing a spider is that it is responsible for finding new work for the indexer. When it finds new documents or finds that documents currently indexed have changed, the content of those documents must be downloaded by the indexer and incorporated in the index. Similarly if documents currently indexed no longer exist, the indexer must be informed so that the index can be updated.

The spider is a program running on the indexer's processor. It finds new work in two main ways. Firstly it recursively follows hyperlinks in known documents to find unknown documents. In the initial stages of spidering, seed URLs are given to the spider and the hyperlink structure is used to populate the index. In later stages, when documents are added or modified, the hyperlinks within these documents can be followed in a similar fashion. Secondly, the spider tracks changes and deletions in documents already indexed, by requesting header information and checking document time-stamps.

Almost all current web spiders work exclusively using HTTP, although protocols for more efficient spidering have been proposed (such as in Harvest). The spider acts as a client conditionally requesting pages from the web information servers in the space of interest. Using HTTP for retrieving documents and following their hyperlink structure is quite efficient, because that is what the protocol was designed for. However, tracking changes and deletions through HTTP is less efficient, because it is based on a "data-pull" communication model and because each request is in terms of a single document. If there is new work for an indexer to do, the only way to find out through HTTP is to send a request for information on *each* document indexed. For large web indexes this means up to 50 million requests before all documents have been checked, only a very small fraction of which will result in new work.

One way of addressing this inefficiency is to reduce the number of HTTP requests a spider must send to find new work. A notable proposal is for a `sitelist.txt` standard, where it becomes each web server's responsibility to provide a single text file listing all files and their time-stamps. However, in such a system the amount of communication is still great regardless of the amount of new work, if any.

2.2 Data Pull vs Data Push

An more efficient method in terms of communication is to abandon the "data-pull" model and rely on the web server to "push" notification of new work back to the indexer. Instead of 50 million largely useless requests or one large request per site, a small notification message is sent at an appropriate time, describing in a concise fashion all new work. Novel server push methods have already been proposed using, for example, a request for email notification whenever a stated document changes.

A "data-push" model for finding new indexing work requires a certain amount of computation and state storage on the web server end. This makes it an ideal application for aglet technology, because not only can this remote computation take place, but the techniques used are determined by the indexer, so new technologies or indexing priorities can be reflected in new versions of the aglet software.

An aglet would be dispatched by the spider/indexer to a web server, acting as its agent, working on behalf of the spider to access and perhaps predigest local information changes at the server, then sending them to the indexer.

2.3 Search industry state of play

Due to the very large volume of documents available on the web, large web search services are expensive to run, both in terms of the cost of spidering and the cost of index building and searching. Because of this, only a small number of larger web indexes exist, and these are all commercial services (AltaVista, Excite, HotBot, Lycos and InfoSeek).

Even these services are forced to compromise for efficiency, only partially covering each site and polling documents infrequently. In fact in some services it is possible for an index to be more than three months out of date with respect to changes in a particular document.

If the cost of spidering is reduced, the overall cost of running a search service also decreases and the coverage of search services may increase. For this reason there is much industry enthusiasm for finding new ways of efficiently detecting change in remote documents. The solution suggested here, a "data-push" model employing aglets, would offer indexers not only very efficient change notification, but could be used in the transmission of the documents themselves, by compressing, pre-indexing or even sending only changed portions of documents.

3. Aglets

3.1 What is an Aglet?

An aglet is a Java-based mobile software agent. The term software agent has been given many definitions; here we refer to a piece of software that can halt its execution on one host, transfer to another host, and then continue execution from where it left off on the remote host.

The aglet framework was created by a team lead by Danny B. Lange at the IBM Tokyo Research Laboratory in Japan.

3.2 Comparing Aglets to Applets

The concept of network-mobile code is most widely demonstrated through web-based Java applets. The client pulls down an applet class file as part of a HTML page. Aglets can be similarly shipped across a network. However, unlike current applets, an aglet’s state is preserved in the transfer. Further, aglets have a degree of *autonomy*; they can control their own migration, deciding when and where to go on a network.

A Java aglet is similar to an applet in that it requires a cogniscant environment in which to run. An applet is executed on a Java application built in to the client browser. Aglets similarly require an aglet host application (known as the Aglet Workbench) to be running on a node before they can visit.

The host application provides a ‘‘sandbox’’ environment for the aglet, enforcing security policies and limiting access to host services.

3.3 The Aglet programming model

Aglets are designed around an event-driven callback programming model that has similarities with the Java Applet programming model. An aglet can experience any of the following events in its life:

Creation: An aglet is instantiated, and its main thread begins executing.

Disposal: An aglet is destroyed, all information is lost.

Cloning: The aglet is replicated, with current state but new identity.

Dispatch: The aglet and state is sent to a remote host.

Retract: A previously dispatched aglet is pulled back from a remote host.

Deactivation: Aglet and its state are transfered to persistent storage.

Activation: Aglet and its state are transfered from persistent storage.

Before any of these events occur, an aglet is notified of the upcoming event through a call to the appropriate callback method. For example, when an aglet is created, the `OnCreation()` method is invoked. A programmer can override this method with one that initialises the state of the newly formed aglet.

3.4 How Aglets move between hosts

Aglets are moved from host to host using the JDK's Object Serialisation feature. The aglet object, all serialisable objects reachable from it, and the aglet's heap are converted into a byte stream and sent across the network. The receiving host can then reconstruct the aglet and its heap.

Java does not allow access to execution stacks of the virtual machine, and so not all state is preserved in the transition. However, state can be effectively restored if the aglet is programmed like a finite state machine. Before dispatch, the current ‘‘state’’ can be recorded in variables on the heap, so then when execution begins at the receiving node, the state variables on the heap can be consulted to determine what to do next.

3.5 The Aglet environment

An aglet can communicate with its host environment via an `AgletContext` object. This object provides access to host services and resources, such as the local filesystem.

An aglet can communicate with another (local or remote) aglet by obtaining an `AgletProxy` object for that aglet. The AgletProxy acts as an intermediary between the aglets, protecting each from making malicous calls to the others public callback functions. The Agletproxy allows an aglet to request an action of another aglet, such as `dispatch()`, or `clone()`. The receiving aglet can choose to carry out the request, ignore it, or take some other action. AgletProxies also allow simple object based message passing between aglets.

3.6 Issues

Security is an important issue in mobile agent technologies, since such technology potentially provides an easy method to propagate malicious code. Conversely, aglets themselves may carry sensitive information, and so aglets must be protected from their host environments. The current aglets framework goes some way to addressing these needs by providing a mulitlayered approach to security. The first layer is provided by the Java Virtual Machine itself. Incoming aglet code is subject to byte code verification before execution. The second layer of security is provided by the security manager, which allows customisable security policies for local and remote aglets in regard to host resources. The final layer is Java's new Security API, which provides services such as encryption, authentification, and digital signatures. The framework is still vulnerable to denial of service style attacks.

IBM is keen to see the aglet model adopted as an open network standard for mobile agents. To this extent they have submitted the aglet framework and transport specifications to the OMG as proposal for the OMG's *Mobile Agent Facility*.

4. Aglets and Spiders

Aglets suggest themselves as an important part of the solution to the problem of the scalability of web spidering for a number of reasons:

- support of a "data push" model should decrease the network traffic and wall time required to locate changes in the web
- the possibility exists for performance-related contracts to be agreed between data suppliers and indexers to ensure that document changes are reflected in indexes in a timely, or at least predicatable, manner (opportunities for contractual consideration exist on both sides - the data provider pays in local computational power, and the indexing site guarantees index presence)
- the functionality of the aglet code is under the control of the indexing site, and so can be updated to reflect its requirements, and its core knowledge of the indexing task
- the binding of an aglet into a local site requires the consent of that site, but in this application such consent is likely (unlike, say, aglets produced by individual persons and dispatched for personal business)
- the spidering problem is fundamentally a graph-traversal problem - the web can be seen as a graph with many alternative arcs (paths) between nodes with different cost-related weights. The ability of aglets to clone, to have an initial intinerary, and to react to local conditions, permits numerous alternative traversal strategies to be explored.

5. References

1. Lange, Danny B. Chang, Daniel T. *IBM Aglets Workbench White Paper* http://www.trl.ibm.co.jp/aglets/whitepaper.html
2. Lange, Danny B. *Java Aglet Application Programming Interface (J-AAPI) White Paper* http://www.trl.ibm.co.jp/aglets/JAAPI-whitepaper.html
3. Venners, Bill *Under the Hood: The architecture of aglets* http://www.javaworld.com/javaworld/jw-04-1997/jw-04-hood.html
4. ALTAVISTA. http://altavista.digital.com
5. Shocked by search engine indexing http://www5.zdnet.com/anchordesk/talkback/talkback_11638.html
6. AltaVista CTO Responds http://www5.zdnet.com/anchordesk/talkback/talkback_13066.html
7. BOWMAN, C. M., DANZIG, P. B., HARDY, D. R., MANBER, U., SCHWARTZ, M. F., AND WESSELS, D. P. Harvest: A scalable, customizable discovery and access system. Technical Report CU-CS-732-94, University of Colorado, Boulder, Colorado, Mar. 1995.
8. The Web Robots Pages http://info.webcrawler.com/mak/projects/robots/robots.html