A Distributed Content-Based Search Engine Based on Mobile Code

Volker Roth
OGM Laboratory LLC
USA
volker.roth@acm.org

Ulrich Pinsdorf
Fraunhofer IGD
Germany
ulrich.pinsdorf@igd.fhg.de

Jan Peters
Fraunhofer IGD
Germany
jan.peters@igd.fhg.de

ABSTRACT

Current search engines crawl the Web, download content, and digest this content locally. For multimedia content, this involves considerable volumes of data. Furthermore, this process covers only publicly available content because content providers are concerned that they otherwise loose control over the distribution of their intellectual property. We present the prototype of our secure and distributed search engine, which dynamically pushes content based feature extraction to image providers. Thereby, the volume of data that is transported over the network is significantly reduced, and the concerns mentioned above are alleviated. The distribution of feature extraction and matching algorithms is done by mobile software agents. We give a description of the search engine's architecture and implementation, quantitative evaluation results, and a discussion of related security mechanism for content protection and server security.

Keywords

H.3.3 [Information Search and Retrieval] Retrieval models, Search process; H.3.4 [Systems and Software] Distributed systems, Information networks; H.3.7 [Digital Libraries]; Content based retrieval, mobile agents, images, content security

1. INTRODUCTION

The availability of vast amounts of multimedia contents in the Internet requires sophisticated means for searching and retrieval. Current search engines are generally based on a centralized gatherer which traverses the hyperlinks of the World Wide Web starting from known entry points, and which retrieves and digests all relevant data found. This approach has two disadvantages: (a) it is data intensive, and (b) search engines cover only contents which are freely available for download. One might argue that transfer volume is not an issue because ample bandwidth is available on the Internet backbones. However, edge networks generally pay considerable penalties if they exceed their transfer volume quotas. They have an interest not to exceed their quota and to keep it as low as possible. Search engines should be designed to honor this desire.

The second disadvantage results from the fact that commercial content providers loose control over the distribution of their intellectual property, once the search engine downloads it. Consequently, providers offer local searches and the number of algorithms they provide is likely limited.

In this paper we report on our distributed search engine prototype, which pushes content-based feature extraction (and optionally feature comparison) to the edge networks, and which can alleviate the aforementioned intellectual property concerns. The search engine is based on *mobile software agents* (see *e.g.* [1] for an introduction to mobile agents). The benefits of mobile agent based feature extraction are:

- Multiple image sources can be processed in parallel. Each image source contributes to the processing power required to extract salient image features of its images.
- Multiple (*e.g.,* composable) feature extraction and comparison algorithms can be deployed concurrently and easily.
- Feature vectors are generally more compact than images, therefore less data must be transported from image sources to the gatherer.
- Feature extraction takes place at the image source. The images must not be exported from it, and original images generally cannot be reproduced from the feature vectors.

The achievable amount of parallelization and the reduced data transfer volumes have a significant positive impact on completion time of the feature extraction process. Disadvantages of the mobile agent based search engine are:

- Image sources have to set aside computing resources for feature extracting agents.
- Running mobile code on a server poses a considerable security risk. Therefore, security of the mobile agent middleware is an essential requirement for the practicality of the approach.

We built our search engine prototype on the mobile agent server *SeMoA*[1] [2]. Although SeMoA supports a rich set of security features, we do not claim that its security is perfect – the fact that it is programmed in Java alone renders it vulnerable to a variety of *Denial of Service* (DoS) attacks [3]. However, SeMoA provides a rich set of cryptographic features to protect the data of agents (*e.g.,* collected images) against disclosure on untrusted hosts, and an architecture that emphasizes separation of agents.

The basic concepts of mobile agent based image search engines have been mentioned by several authors before [4, 5, 6, 7] but have not yet been addressed in sufficient detail and in the context of a

---

[1]See *e.g.,* http://www.semoa.org

practical system. In this paper, we contribute a nuanced discussion and comparison of operation modes, a description of our implementation, and a quantitative analysis of the benefits of our search engine.

2. CONCEPTS AND MODELS

There does not seem to be a universal understanding when a program crosses the border to agent-hood. Software agents are often defined as being *reactive, autonomous, goal-oriented,* and *continuous* [8] though further attributes exist. *Mobile* agents have the ability to relocate – at some point of their execution they can halt and initiate a *migration* to some distant host where they resume execution. During migration, an agent's program as well as its current execution state and accompanying data is transported to its new host. When and where an agent migrates is part of the agent's program. In general, mobile agents rely on an infrastructure of mobile agent servers which handle agent transport, setup and deinstallation. What mobile agent technology brings to bear on the problem of image indexing and retrieval is easy means of *software distribution*. Briefly, mobile agents provide a flexible and easy mechanism to transport content-based feature extraction and matching algorithms to the source of the images rather than vice versa. The impact on network utilization and scalability if profound – instead of putting the burden of gathering contents completely onto the shoulders of a centralized gatherer and its connected network interface the load is shared among the gatherer and the image servers and all image servers can be indexed in parallel.

Below, we illustrate two slightly varying models of image search engines based on mobile agent and content-based retrieval technology. The first model resembles an optimized gatherer which still has a central repository of feature vectors (though feature extraction is done remotely). We refer to this model as the *gatherer model*. The second model keeps a distributed index and no data needs to be shipped over the network during indexing. We refer to this model as the *incubator model*.[2] Below, we describe both models. In §3.1, we explain in greater detail to what extent and how we have implemented the models in our prototype.

2.1 The Gatherer Model

The gatherer model consists of a central image broker, several image servers, *index agents, search agents,* and *fetch agents* (all agents are mobile and relocate during their life cycle). The image broker dispatches index agents which transport feature extraction algorithms to one or more image servers. On these servers, the index agents extract relevant feature vectors from local images and send or take image entries back to the image broker. At the image broker, all image entries are merged into the central index (see Fig. 1 for illustration). Each image entry consists of a feature vector, the URL pointing to the host where the image was retrieved, an image ID that uniquely identifies the image at the image server, an optional thumbnail, and optional further information on the image such as its size. The globally unique ID of an image consists of the globally unique URL plus the locally unqiue image ID.

Based on the index data, image brokers can either serve requests in a client/server fashion, or they support mobile agent queries as follows. A client sends a search agent to the broker which queries for similar images by means of an example image, a sketch, a prototypical image, or a feature vector which is extracted from either of these query images. The query result consists of extended image entries which contain the normalized distance between the query and the entry's feature vector in addition to the entry itself. The search agent transports the result set back to the client who selects images for retrieval based on the included thumbnails, or refines the query (*e.g.,* by means of relevance feedback). Once an image is selected for retrieval, the client sends a fetch agent that migrates to the server on which the image is stored (directed by the URL which is stored in the image entry) and retrieves the image based on the local image ID also contained in the entry (see Fig. 1). This can be preceded by a negotiation phase in which agent and image provider agree *e.g.,* on licensing terms and the payment of license fees.
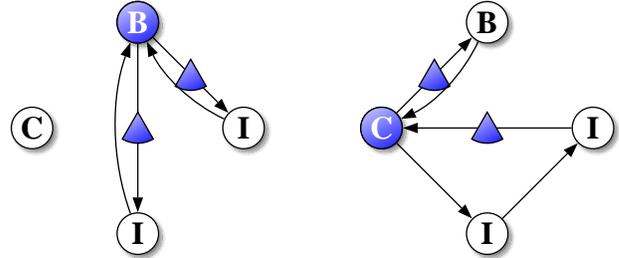

Figure 1: (Gatherer model) Left half: the broker (B) dispatches feature extraction agents (denoted as triangles) to the image sources (I). Once the agents completed extracting the features of all images, they carry the feature vectors back to the broker. Right half: The client (C) sends a search agent to the broker (B), which retrieves image entries of similar images, and transports the entries back to the client. The client selects images for retrieval, which are subsequently collected from the image sources (I) by the fetch agent.

The transfer volume savings of this model are proportional to the compression factor of the feature extraction algorithm. A constant overhead incurs because the feature extraction algorithm must be transported to each image server. On the other hand, only one network connection request is required for transporting the agent compared to one connection per image in the case of ordinary gatherers (unless the gatherer and the image server support sessions). Content providers retain control over their intellectual property because only feature vectors are exported, from which the original image generally cannot be reproduced in high quality.

2.2 The Incubator Model

The incubator model can do even better than the optimized gatherer model. In the incubator model, one index agent per image server is dispatched and takes residence at the image server. There, it extracts features as previously described but sets up an index directly at the image server (see Fig. 2). The index agent may also monitor the local image repository for changes, and it can update its index accordingly and incrementally. The only communication between the broker and the index agent is a short notice that the computation of the index is completed and the index agent is ready to provide service to search agents. The image broker is still a central point of access but it resembles more a yellow page server. It refers search agents to the image servers where index agents reside (see Fig. 2) Once the client selects an image for retrieval, the process continues as in the gatherer model.

Based on its index the index agent serves queries of search agents which visit the image server. On each image server, the search agent merges previously collected results with the results of its local search, and prunes the overall number of results *e.g.,* to a user-defined maximum number of n images with the lowest overall distance metric (unless security considerations take precedence,

---

[2]Incubator: "A place or situation that permits or encourages the formation and development, as of new ideas." The American Heritage Dictionary of the English Language, Fourth Edition
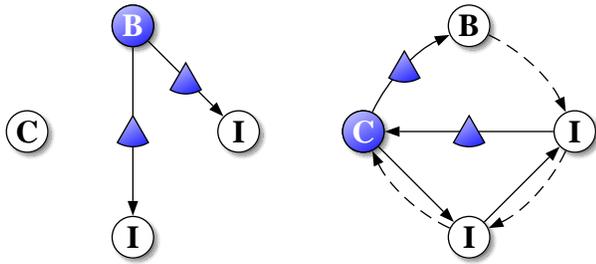
Figure 2: (Incubator model) Left half: the broker (B) dispatches feature extraction agents (denoted as triangles) to the image sources (I). Once the agents completed extracting the features of all images, they register a feature comparison service at the image sources. Right half: the client (C) sends a search agent to the broker (B), which retrieves a list of image sources, and searches them in turn (dashed lines). The client selects images for retrieval, which are subsequently collected from the image sources (I) by the fetch agent.

see also §3.2). The distance metric must be normalized so that the pruning is accurate. This approach is simple to implement. Although, the search agent has to carry n images along with it. On the other hand, multiple search agents can be dispatched in parallel to speed up the search. In such a case, the individual search results must be merged and pruned at a suitable host *e.g.* a server provided by the broker or the client itself.

Additionally, brokers may launch search agents on behalf of a client *e.g.,* if clients do not use mobile agents directly but rather access the broker through a regular Web interface.

2.3 Comparison of Concepts and Benefits

The advantage of the gatherer and the incubator model over traditional centralized image repositories is that processor and memory consumption is shared between the image providers whose contents are processed, and the broker. Network utilization is considerably lower in both mobile agent based models than in the traditional approach, and the feature extraction process completes considerably faster as a consequence of parallelization. None of the two models export image contents to third parties.

The gatherer model is still somewhat centralized. Queries are answered by the broker. If the number of images is huge then the feature collection is likewise huge. Hence, while the gatherer model improves the process of index compilation it does not significantly improve the query process. Finally, if the broker fails then the entire service becomes unavailable.

Searching is less efficient in the incubator model than it is in the gatherer model because all image servers must be visited by the search agent in turn before the query results are shown to the user. Although this can be alleviated by sending multiple search agents in parallel. However, the yellow pages maintained by the broker are much smaller than a full index of feature vectors, and they change less often then an index.[3] Therefore, replication (*e.g.,* by caching or by fail-over servers) can be implemented easier and more efficiently than this would be possible in the gatherer model.

The incubator model is particularly useful if image providers (who offer a broad range of images) team up with image brokers (who distribute specialized retrieval algorithms). Hence, the image broker may act as a well-known *portal site* with a focused marketing that addresses a specific target audience. The relationship be-

tween providers and brokers can be many-to-many, and their business relationships can be fluent and flexible. The advantage is that both parties can concentrate on their core competencies. Image provider specializes on content provisioning, and the image broker specializes on retrieval technology and retrieval services. In this regard our approach differs from *e.g.* meta search engines, which combine the results of regular search engines that in turn download contents. In our approach, however, search functionality is pushed to the content.

Both models (the gatherer and the incubator model) can support multiple content-based retrieval mechanisms in parallel as well as retrieval mechanisms based on annotated information such as the name of photographer or painter, the year of production, or the price of licensing the image for specific uses). For instance, image servers can accept more than one index agent at the same time, and search agents can compute the intersection of multiple distinct result sets based on the global or local ID of each image entry.

3. ARCHITECTURE

Both the gatherer model and the incubator model are built on top of the SeMoA mobile code middleware [2]. Agents are received by network transport daemons, and they are injected into a pipeline of filters (see also Fig. 3) which perform various security services and security checks on incoming agents (see §3.2 for more details). If an agent is admitted to the server then the agent may publish or retrieve service interface objects subject to access control restrictions. Upon termination, the agent is again processed by a filter pipeline and is subsequently migrated to its next hop.
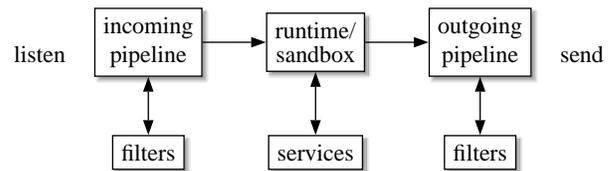

Figure 3: The middleware runs a daemon which listens for incoming agents. Each agent is piped through several filters before it is admitted to the runtime system where it can access services by name. Agents can register and retrieve services by name (subject to access control). Before migration, each agent is piped through outgoing filters again.

Services are published in a hierarchical name space (similar to a hierarchical file system), which simplifies the grouping of services and the definition of access control policies. Agents may publish services dynamically at runtime in an allowed subspace of the name space, and the server may publish services or launch daemons statically at boot time. For instance, an image broker provides a static *index* service that his agents (and only his agents) can access in order to merge collected feature vectors with previously collected ones. In our implementation, the *index* service is backed by a file system and provides concurrent reader/writer access to the stored information. The image broker also publishes a static *finder* service which, on input of a query, returns matching image entries. This service is backed by the *index* service (as illustrated by the horizontal arrow in Fig. 4) but restricts access to the index to a limited set of operations and can therefore be made accessible to search agents (by placing it in the public area of the name space). In the incubator model, an index agent publishes the *finder* service dynamically at the image provider, and it keeps a private (unpublished) *index* service as the back end of its *finder* service. In that case, the image entries are stored by which ever resource backs the

---

[3]Image providers regularly add content but if a server is added or removed from the system then content is added or removed as well.
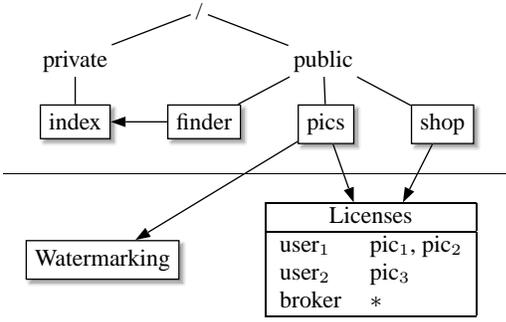
Figure 4: Agents can publish and retrieve services in a hierarchical name space. For instance, an image provider publishes the *pics* service under the path "/public/pics". The *pics* service iterates image names and thumbnails without restriction, but retrieves full quality images only if it is invoked by an agent whose owner has purchased a license. Agents can negotiate and purchase licenses on behalf of their owners by the *shop* service.
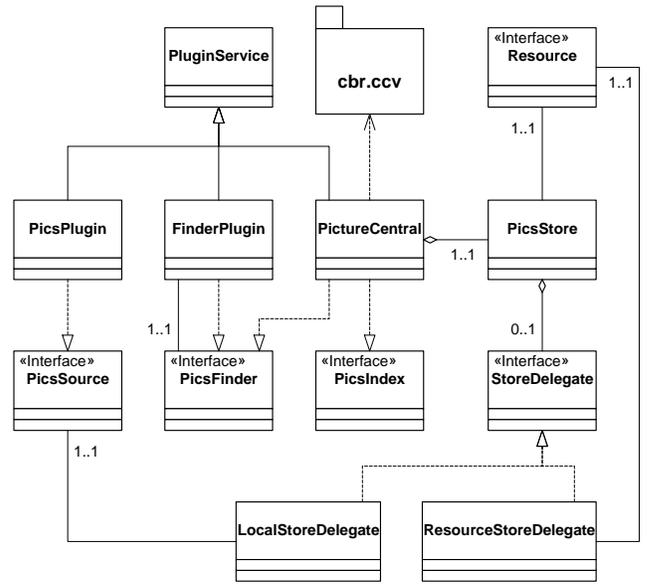

Figure 5: A simplified view of the interface and class design related to the CBR services is given above. Storage of information is handled through the "Store" abstraction for which we implementated instances that map to RAM or file systems.

storage of the index agent (typically a file system or RAM of the host computer, the middleware provides abstractions for the actual type of agent storage which could also be backed by a database). Figure 5 gives a simplified view of the interface and class design in the UML [9] notation.

Image providers publish the static *pics* and *shop* services. The *pics* service iterates image IDs (*e.g.,* a locally unique image name) and thumbnails without restriction, but retrieves full quality images (based on the image ID) only if the owner of the invoking agent already purchased a license for the image in question. In this case, the *pics* service may also embed the client ID as a digital watermark[4] into the retrieved image so that unauthorized usage of copyrighted material can be traced. Clients (resp. their agents) can negotiate and purchase licenses by the *shop* service. (While the *shop* service and corresponding license verification is a conceptual component of our architecture we have not yet engineered it.) Brokers can purchase a license for all images for the purpose of indexing (presumably at a low price and under the legally important condition that no full quality images are illicitly exported). Alternatively, image providers can grant brokers access to their images based on prior offline agreement.

The *pics* service provides a simple and sufficient interface so that feature extraction algorithms can iterate and extract features from existing images, irrespectable of the heterogeneity of deployed image databases (*e.g.,* the schema of the database or the fact that images are simply stored in a file system).

3.1 Implementation Details

The prototype implementation uses *Color Coherence Vectors* [11] as feature extraction and comparison algorithm. Feature vectors consist of 128 float values; each vector is computed as follows: the image is blurred using a simple 3×3 convolution filter which averages the color values of all horizontal and vertical neighbors of the filtered pixel. The blurred image is then quantized to a color space of 64 colors. In the last step, the pixels of the image are classified into coherent and incoherent pixels. Coherent pixels

are pixels which are part of a horizontally and vertically connected pixel area of the same color whose size exceeds a certain threshold τ (a fixed percentage of the total image area). Incoherent pixels are pixels which are not coherent pixels. For each of the 64 colors, the coherent and incoherent pixel counts are summed up separately and normalized with regard to the total image area. This results in a 128 dimensional vector. The L_1 distance is taken as a measure of similarity between two color coherence vectors. Let $\langle (h_1, \bar{h}_1), \ldots, (h_n, \bar{h}_n) \rangle$ be a color coherence vector where h_i is the percentage of coherent pixels of color i and $\bar{h}_i$ is the percentage of incoherent pixels of color i then the L_1 distance is defined as: $\|h - h'\| = \sum_{i=1}^{n} (\mid h_i - h'_i \mid + \mid \bar{h}_i - \bar{h}'_i \mid)$. The Color Coherence Vector algorithm has the advantage that it is easy to implement, reasonably fast, and achieves a high compression rate.

Images are reduced to a vector whose encoding is less than 600 bytes. For feature extraction algorithms whose output has a length comparable to the size of the images no volume transfer savings are achieved. Although in this case the processor utilization is still distributed among the image servers (this results in a speedup linear in the number of image servers, given a uniform distribution of images). Here, we assume that *number of search engines ≪ number of image servers*.

The graphical user interface of the demonstrator is shown in Fig. 6. The left view shows the panel which is used to launch index agents. From a list of known servers, a subset can be chosen. On pressing the button titled *start indexing*, index agents are created and dispatched to each selected server. On the target server, each index agent looks up the *pics* service which must be registered in the target server, and starts the feature extraction process. Upon completion, it publishes an instance of the *finder* service in the target server, which search agents can look up and query.

The middle view shows the panel which is used to dispatch search agents. Again, a number of servers can be chosen from a given list. The search agent takes a user-provided example image (which can as well be a sketch or prototypical image), hops in turn

---

[4]Put simply, the term "digital watermarking" refers to steganographic means of embedding copyright markers in multimedia data so that the marking is imperceptible, undetachable, as well as robust against a variety of adverse and inadvertent manipulations of the media such as lossy compression, format conversion, *et cetera*. See *e.g.,* [10] for an overview over digital watermarking.

to all selected image servers, and collects image entries with a distance less than a given threshold and up to a given maximum number. The overall best matches (thumbnails but not full images) are reported back and presented in the results panel.

The results panel shows the retrieved thumbnails and each entry's distance to the query image. If the user clicks on a thumbnail then a fetch agent is created and dispatched to the host where the image was retrieved, and returns the corresponding full image. The check box in the lower left corner of the results panel enables secure retrieval. If it is checked then retrieved images are transported in encrypted form as explained in [12].

Although we implemented only one feature extracting and matching method so far, the interfaces are designed to support multiple and alternative implementations transparently. All implementations have been developed in the Java programming language (Java Version 2).

3.2 Content Protection

It has been argued that mobile agents achieve a greater level of control over the media being searched on [4]. This is only part of the truth, though. In practice, various covert channels [13] as well as direct means of cheating can be used *e.g.,* by malicious index agents and colluding search agents to subvert image export restrictions. The billing schemes proposed by Belmon and Yee (which account for projected losses due to covert channels) punish thieves and ordinary clients likewise and will hardly be accepted. Still, using *e.g.,* the incubator model can improve confidence that image contents are not exported illicitly from image servers. Evidence of stealing images on the part of index agents can be established by reverse engineering the agents' code, if it comes to the worst. SeMoA requires that each sender of an agent digitally signs the static parts of his agent (including the code), which establishes a non-repudiable proof of ownership. This signature yields a unique and unforgeable agent *kernel.* Furthermore, each server must sign the entire agent before transport. This signature binds the new state of the agent to its kernel and protects the agent against tampering during transport. Thereby each server documents its responsibility for any state changes that the agent may have undergone while being hosted by it (see Fig. 7 for an illustration of signatures).

Nevertheless, an agent must also have means of protecting itself against a malicious host as soon as for-profit services are involved. If search agents pass by multiple competing image providers there is a certain chance that a provider does not play by the rules. One possible attack that a dishonest provider may launch on an agent is to manipulate the accumulated search results or to replace or degrade the quality of images retrieved from a competitor. Thereby the perpetrator increases the likelihood that images are purchased from him the next time. Other attacks involve tampering with the agent's code in order to alter its negotiation strategy, decision making, or simply to cause harm at the servers of competitors.

SeMoA prevents tampering with the code by requiring that code must be digitally signed. Any such tampering invalidates the kernel of the agent and hence the agent subsequently fails to "speak on behalf of its original owner." Tampering with accumulated images is prevented by establishing encrypted subsections in the agent so that each subsection can be accessed only by the agent's owner and by hosts that he authorizes [12]. For instance, the fetch agent is programmed so that it stores each retrieved image in the section that is dedicated to its current host. This section is automatically encrypted by the *encrypt* filter when the agent departs.

This setup has a drawback, though. Agents cannot access results that they collected prior to reaching the current host (these results are encrypted and by assumption the current host does not have
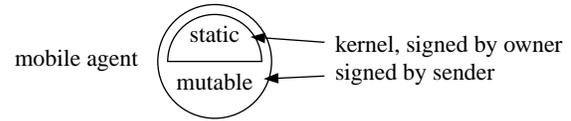

Figure 7: A mobile agent consists of static data (which does not change during the agent's lifetime *e.g.,* code and a random agent ID) and mutable state. The owner assumes responsibility for her agent by signing its static part (the *kernel*), and the most recent host of an agent signs the entire agent to assume responsibility for the agent's most recent state.
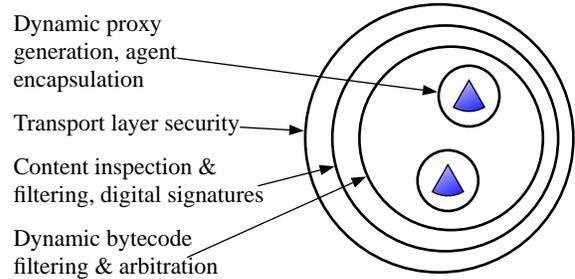

Figure 8: The security architecture resembles an onion. Agents have to pass all layers successfully to be admitted to the system. The outer layers keep threats out of the system. The innermost layer encapsulates and confines agents.

a matching private key). This prevents an agent from pruning its accumulated results *e.g.,* to a fixed upper number of overall best results (we referred to this problem in §2.2). One workaround is to program agents so that they regularly migrate to a trusted and authorized host where the agent can access and prune all results that have been accumulated up to this point.

3.3 Server Safety and Security

For practical purposes it is essential that agent servers are protected against attacks by malicious agents. Agents must not be able to disrupt or otherwise negatively effect the operation of an image server or other agents hosted by it. For mobile agent servers based on Java this is currently an elusive goal – the Java runtime system is vulnerable to a variety of *denial of service* attacks [3].

However, SeMoA makes a best effort to protect the runtime system against malicious code, and provides pluggable bytecode filtering and arbitration modules. Before a class is loaded into the name space of an agent each module may inspect, reject, or instrument the bytecode of that class. Currently, SeMoA includes a module that rejects classes which *e.g.,* override the `finalize` method, a well-known and simple way to attack the garbage collector thread of the virtual machine (a more subtle variant of this attack may be directed at the `close` method of some I/O classes). The same extension mechanism can be used to add resource control by bytecode arbitration [14] as well as additional security checks.

Each agent is run in a separate name space with a separate class loader and in a separate thread group. Thereby, interference of agents prevented. A special security manager filters and sorts newly created threads so that for instance threads of the *Abstract Window Toolkit* are not accidentally placed in the thread group of an agent. Marshalling and unmarshalling is done by the initial agent thread from within the agent's sandbox so that an agent cannot exploit callbacks in the *Java Serialization Framework* to hijack server threads. The server transports an agent only after all threads of that
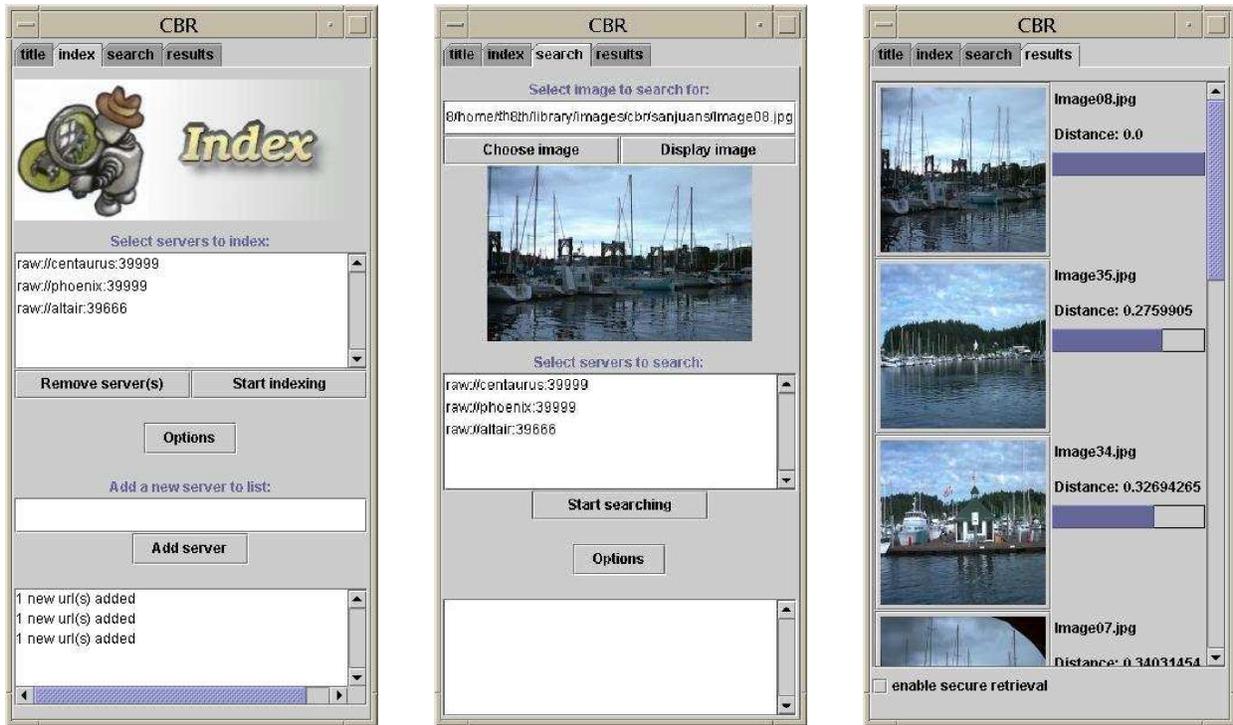
Figure 6: Three shots of the prototype GUI. The panel used to launch index agents is left, in the middle is the panel used to start search agents for a given example query image, and the panel which shows the query results after the search agent returned is right.

agent have terminated, which prevents the adverse or inadvertent creation of clones or zombies (or, for that matter, widespread infection of servers by a worm).

Once running, an agent may publish and retrieve service objects (such as the *finder* service) by name in the server's object registry if the agent has appropriate permissions. Published objects are automatically wrapped in a proxy object which is created dynamically. The proxy prevents uncontrolled aliasing of the service object and automatically invalidates references to it as soon as the agent terminates. This makes the service object available for garbage collection. Agents cannot share classes (by virtue of separate name spaces the classes would not be type-compatible) but they may share interfaces for the purpose of method invocation and communication. However, two interfaces are shared across name spaces only if their implementations are mapped to the same image by a cryptographic hash function. In such a case, a superordinate class loader assures type-compatibility.

Before an agent is loaded and run, it must pass a configurable pipeline of pluggable filter modules (see also Fig. 3). Each filter may reject the agent in the case of errors. SeMoA provides a variety of security related filters some of which transparently handle digital signatures, certificate chain validation, and encryption of subsections of an agent. Agent transport is possible both in the clear and over a mutually authenticated *Secure Sockets Layer* (SSL) connection. A schematic illustration of SeMoA's security architecture is given in Fig. 8.

4. EVALUATION

In the incubator model, no feature vectors must ever be transported over the network. Therefore, a quantitative evaluation of this model is superfluous. We evaluated the gatherer model of our prototype for small sets of images in order to get a general idea of

the potential savings that can be achieved by mobile agents compared to conventional client/server approaches. The setup of the experiments favored the conventional approach so that our results remain conservative. In practice, we expect that the relative performance of mobile agents is better. We used the hardware and software given below in our evaluation:

- $1 \times$ Pentium III mobile, 1.2 GHz, 512 MB, FreeBSD 4.7, running Java Version 1.4.1 under the Linux emulation.
- $9 \times$ Sun Ultra 5/10, 500 Mhz (UltraSPARC-IIe), SunOS 5.8/5.9, 256MB-512MB, running Java Version 1.4.1, *Apache* Server Version 1.3.
- Switched Fast Ethernet (100 Mbit/s)

The nine computers we used were connected by our institute's LAN, which consists of several hundred workstations and PCs, and which is accessed by more than 150 research assistants and countless students (although we did our tests a weekend to reduce distortion of measurements due to regular use of the network).

We first measured the performance of the conventional approach. A simple client program loaded and extracted the features of all images, with a varying number of image sources. The client was programmed in the Java programming language; it ran on the 1.2 GHz Pentium III laptop, it used three threads per image source in parallel to optimize I/O utilization (we found by experimenting that this gave the best results), and it was based on the same code that was used by the mobile agents in subsequent testing.

The image sources consisted of 8 Apache servers, each of which ran an Apache server with 48 images. All images were in JPEG format with a resolution of 756×504 pixels, and were loaded by the Apache server from the built-in hard drive. The size of images varied from approximately 280000 to 456000 bytes, depending on the JPEG compression rate. Each experiment was done three times
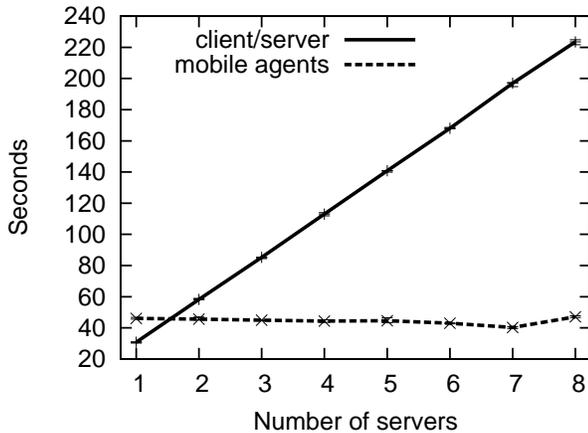
Figure 9: The time that the client/server based gather needs to complete the feature vector collection task vs the time required by the mobile agent approach (measured from starting the first agent until completion of the last agent). The point of intersection is the break-even point at which the additional overhead of shipping and setting up mobile code is amortized by the reduced network utilization of the mobile code approach.
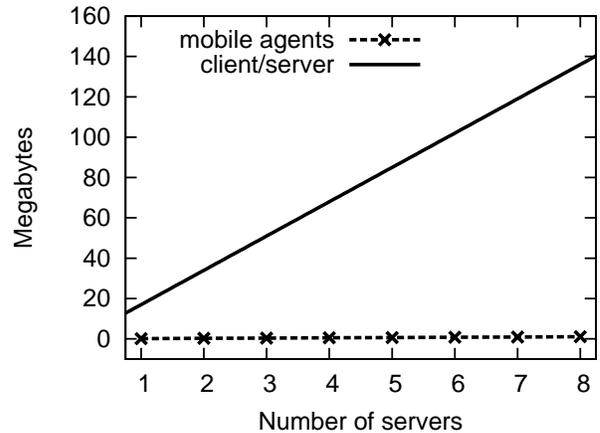

Figure 10: The amount of data transported of the network in the client/server based gatherer vs the mobile agent approach. The client/server graph is plotted based on the size of all downloaded images times number of servers. The graph of the mobile agent approach has a slope which is too small to be noticeable compared to the upper graph.

to observe variances.

In the measurement of the mobile agent performance each Sun hosted a mobile agent server. The ninth Sun (without images) played the role of the broker. The other Suns were configured with a simple service that allows to iterate picture names and to retrieve image data for an image with a given name. Again, all images were loaded from the built-in hard drive.

In experiment one, we launched a mobile agent on the broker. This agent migrated to image server one, extracted the features of all images, transported the image entries back (excluding thumbnails), and merged the image entries with the broker's central index. In the second through eighth experiment, we launched two to eight agents in parallel which performed the same operations on the additional image servers. In each experiment, we measured the time from starting the first agent until the last agent returned and completed its task. Again, we repeated all experiments three times.

We also measured the sums of sizes of all image entries (including overheads for the agents) transported per experiment. Comparison of all collected results shows significant savings in favor of the mobile agent approach, as could be expected (see Fig. 9 and 10, min and max values deviate so little from the median and mean values of the client/server and mobile agent measurements that the error bars are hardly noticeable in the graphs).

Additionally, our image search engine was the subject of a field study with the objective to assess the legal aspects of electronic commerce with mobile agents. Over the period of two days, eleven lawyers used our retrieval system in the roles of the content provider, customer, and image broker, with the objective to trade images. Overall, more than a thousand agents were dispatched. Our prototype proved to be user-friendly, robust, and reliable.

5. RELATED WORK

Considerable work is done in the general area of CBR, see *e.g.*, [15] for the proceedings of a recent conference. Well received work by several authors reports on CBR systems for the World Wide Web [16, 17, 18]. All these image search engines are based on the client/server paradigm of collecting images from the Web. Mobile agent technology is complementary to this work. It remains to be investigated how well the algorithms developed by the authors mentioned above can be adapted to be used within a mobile agent framework. The idea of using mobile agents for content based image retrieval has been mentioned before [6, 5, 7]. Mobile agents have also been applied in related applications. For instance, in [19], Johansen reports on the use of mobile agents in the context of a weather information system (mobile agents process and deduce weather information from satellite imagery). It is not our intention to claim originality of the idea, but to report unique aspects of our architecture (the incubator and the gatherer model), the results of our evaluation, and our experiences with respect to the usefulness of the application.

6. SUMMARY

Digital images are a valuable commodity and we expect that more and more photo agencies make use of the Internet as the principal platform for advertising, customer relationship management, and – most importantly – content distribution. We presented two models of deploying mobile agents to gather image information from the Internet. Both models take into account that content providers must retain control over their intellectual property. Multiple complimentary retrieval methods can operate in parallel. The models support flexible software distribution, updates, and deinstallation, and they can be extended to account for negotiation of license terms and automatic fingerprinting of retrieved images based on digital watermarks.

The models differ in the grade of decentralization. In the gatherer model, the amount of data transported over the network depends on the size of the images and the compression factor of the deployed feature extraction algorithms. In our case, this is less than 1% of the image data transported by regular gatherers. In the incubator model, no image data is shipped over the network at all. Independently of the size of feature vectors, both models achieve a constant speedup, which is proportional to the number of image servers indexed in parallel. Image providers must set up and reserve

computing resources for the mobile agent server. They can operate this server in conjunction with a Web server *e.g.,* by attaching the agent server to the Web server by Servlets.

One avenue for improvement is the combination of the incubator and the gatherer model. The index agent that takes residence at the image provider may cluster the feature vectors *e.g.,* as described in [20]. Rather than sending only its *finished* message to the broker it may submit a number of centroids of the densest clusters. Search agents which visit the broker may thus opportunistically prune the search space by migrating only to servers with centroids most similar to the query vector.

Mobile agent infrastructures require a sound security model, which accounts for the various threats. Some progress has been achieved in the area of mobile agent security [21, 22], although a number of hard problems are still unsolved. Yet it is probably fair to say that in principle the attainable level of security is reasonable enough to justify the application of mobile agents in some real-world applications. However, before this can happen, runtime systems must become more robust *e.g.,* Java must become considerably more robust against *denial of service* attacks [3].

Acknowledgments

7. REFERENCES

[1] James E. White. Mobile agents. In J. Bradshaw, editor, *Software Agents*, chapter 18, pages 437–472. AAAI/MIT Press, Menlo Park, CA, 1997.

[2] Volker Roth and Mehrdad Jalali. Concepts and architecture of a security-centric mobile agent server. In *Proc. Fifth International Symposium on Autonomous Decentralized Systems (ISADS 2001)*, pages 435–442, Dallas, Texas, U.S.A., March 2001. IEEE Computer Society. ISBN 0-7695-1065-5.

[3] Walter Binder and Volker Roth. Secure mobile agent systems using Java – where are we heading? In *Proc. 17th ACM Symposium on Applied Computing, Special Track on Agents, Interactions, Mobility, and Systems (SAC/AIMS)*, Madrid, Spain, March 2002. ACM.

[4] S. G. Belmon and B. S. Yee. Mobile agents and intellectual property protection. In Rothermel and Hohl [23], pages 172–182.

[5] C. Arora, P. Nirankari, H. Ghosh, and S. Chaudhury. Content based image retrieval using mobile agents. In *Third International Conference on Computational Intelligence and Multimedia Applications (ICCIMA '99)*, pages 248–252, 1999.

[6] Volker Roth. Distributed image indexing and retrieval with mobile agents. In *IEE European Workshop on Distributed Imaging*, number 1999/109 in IEE Electronics & Communications, pages 14/1–14/5, Savoy Place, London, WC2R 0BL, UK, November 1999. IEE. ISSN 0963-3308.

[7] J. You and H. A. Cohen. A new approach to image retrieval by fast indexing and searching. In *Proc. DICTA'97*, pages 425–430, Auckland, N.Z., 1997.

[8] Stan Franklin and Art Graesser. Is it an agent, or just a program? In *Intelligent Agents III*, volume 1193 of *Lecture Notes in Artificial Intelligence*, pages 21–36, Berlin, 1997. Springer Verlag.

[9] Martin Fowler and Kendall Scott. *UML Distilled*. Addison-Wesley, Reading, Massachusetts, U. S. A., 1997.

[10] Stephen Wolthusen Michael Arnold, Martin Schmucker. *Techniques and Applications of Digital Watermarking and Content Protection*. Artech House, 2003. ISBN 1-58053-111-3.

[11] Greg Pass, Ramin Zabih, and Justin Miller. Comparing images using color coherence vectors. In *Proc. ACM Conference on Multimedia*, Boston, Massachusetts, U. S. A., November 1996.

[12] Volker Roth and Vania Conan. Encrypting Java Archives and its application to mobile agent security. In Frank Dignum and Carles Sierra, editors, *Agent Mediated Electronic Commerce: A European Perspective*, volume 1991 of *Lecture Notes in Artifical Intelligence*, pages 232–244. Springer Verlag, Berlin, 2001.

[13] B. W. Lampson. A note on the confinement problem. *Communications of the ACM*, 10:613–615, October 1973.

[14] Walter Binder Jarle G. Hulaas and Alex Villanzon. Portable resource control in Java: Application to mobile agent security. In *1st Int'l Workshop on Secure Mobile Multi-Agent Systems at the 5th Int'l Conference on Autonomous Agents*, Montreal, Canada, May 2001.

[15] M. S. Lew, N. Sebe, and J. P. Eakins, editors. *Proc, Int'l Conference on Image and Video Retrieval*, volume 2383 of *Lecture Notes in Computer Science*. Springer Verlag, July 2002.

[16] J. R. Smith and S.-F. Chang. An image and video search engine for the world–wide web. In *Proc. Storage and Retrieval for Image and Video Databases V (SPIE)*, San Jose, CA, USA, February 1997.

[17] S. Sclaroff, L. Taycher, and M. La Cascia. ImageRover: A content-based image browser for the world wide web. In *Proc. IEEE Workshop on Content-based Access of Image and Video Libraries*, San Juan, Puerto Rico, June 1997.

[18] M. Beigi, A. B. Benitez, and S.-F. Chang. MetaSEEk: A content–based meta–search engine for images. In *Proc. Storage and Retrieval for Image and Video Databases VI (SPIE)*, San Jose, CA, USA, January 1998.

[19] Dag Johansen. Mobile agent applicability. In Rothermel and Hohl [23], pages 80–98.

[20] Stephan Volmer. Buoy Indexing of Metric Feature Spaces for Fast Approximate Image Queries. In *Proc. Eurographics 2001 Workshop on Multimedia*, pages 121–130, Manchester, UK, September 2001.

[21] Giovanni Vigna, editor. *Mobile Agents and Security*, volume 1419 of *Lecture Notes in Computer Science*. Springer Verlag, Berlin Heidelberg, 1998.

[22] Jan Vitek and Christian Jensen. *Secure Internet Programming: Security Issues for Mobile and Distributed Objects*, volume 1603 of *Lecture Notes in Computer Science*. Springer-Verlag Inc., New York, NY, USA, 1999.

[23] K. Rothermel and F. Hohl, editors. *Proceedings of the Second International Workshop on Mobile Agents (MA '98)*, volume 1477 of *Lecture Notes in Computer Science*. Springer Verlag, Berlin Heidelberg, September 1998.