Liquid Software: A New Paradigm for Networked Systems

John Hartman, Udi Manber,
Larry Peterson, and Todd Proebsting

TR 96-11

Abstract

This paper introduces the idea of dynamically moving functionality in a network—between clients and servers, and between hosts at the edge of the network and nodes inside the network. At the heart of moving functionality is the ability to support *mobile code*—code that is not tied to any single machine, but instead can easily move from one machine to another. Mobile code has been studied mostly for application-level code. This paper explores its use for all facets of the network, and in a much more general way. Issues of efficiency, interface design, security, and resource allocation, among others, are addressed. We use the term *liquid software* to describe the complete picture—liquid software is an entire infrastructure for dynamically moving functionality throughout a network. We expect liquid software to enble new paradigms, such as *active networks* that allow users and applications to customize the network by interjecting code into it.

Department of Computer Science
The University of Arizona
Tucson, AZ 85721

1 Introduction

The dramatic success of the World-Wide Web can in large part be attributed to its ability to provide location-independent access to data—with the click of a button a user can access information stored across the country as easily as that stored on his or her own computer. No longer must the user differentiate between data stored locally and that stored remotely and use different tools and methods to access the two; indeed the user need not know where any of the data are actually stored.

Although the benefit of location-independent data access can be measured by the tremendous growth in the Web, the underlying functionality of the Web is relatively static. It supports a finite set of protocols to move data between computers. One's local computer can only manipulate the data in a fixed number of ways, determined by its current software configuration. If a user lacks a particular application needed to properly access or view a data file the user must manually install the needed application. Thus, while location-independent data access is a wonderful thing, it is only the tip of the iceberg of possibilities opened by the Web. One can imagine location-independent (mobile) code—code that is not tied to any particular location in the Web, and whose actual location is not a concern of the users. Code located remotely can be executed as easily as code stored locally. This freedom of code location is the true promise of the Web—users are no longer constrained by the physical boundaries of their computer, neither in the code it can run nor the data it can access.

Location-independent code requires dynamically moving functionality around in a network—between clients and servers, and between hosts at the edge of the network and nodes inside the network. At the heart of moving functionality is the ability to support *mobile code*—code that is not tied to any single machine, but instead can easily move from one machine to another. In addition to mobile code, a system that dynamically moves functionality must also address the efficiency, interface, security, and resource allocation concerns that mobile code raises. We use the term *liquid software* to describe the complete picture—liquid software is an entire infrastructure for dynamically moving functionality.

Making software more liquid seems to have several potential advantages. On the one hand, the ability to move software simplifies many tasks that today are awkward at best, and impossible at worst. For example, liquid software simplifies remote execution since it gives one site the ability download the modules it needs to access the resources at another site. Remote execution, in turn, makes it possible to do software installation, diagnostics, and maintenance at a distance. As another example, liquid software facilitates seamless updates, thereby supporting just-in-time downloading of software updates, as well as the evolution of software over time. As a final example, liquid software makes it possible for clients (users) to customize the service provided by different servers they contact.

On the other hand, if instead of thinking in terms of how the ability to move code makes things we do today easier we start from first principles and ask ourselves what functionality is and is not tied to a particular location, then we can imagine countless ways in which we can exploit liquid software. For example, while a certain number of cycles have to be available at the end-user's host—to execute the display-specific aspects of the GUI and to encrypt/decrypt data that is sent over the network—most information is not location-dependent. The predominate reason information is today tied to a particular location (machine) is that we cannot flexibly move the code that can manipulates the data to some other location.

Our approach to developing liquid software as a paradigm for networked systems has three major thrusts. First, we are developing an enabling technology—fast compilation of machine-independent code—that will facilitate liquid software. Second, we are implementing an application—mobile searching—that can benefit from mobile code, and use it as a driving force in exploring the viability of liquid software. Finally, we are building a demonstration system that integrates our compiler technology and our searching application on an OS platform that is tailored to support liquid software. This demonstration will necessarily require that we flesh out a complete system infrastructure that supports liquid software.

2 Gigabit Compiler

Liquid software presents research challenges that naturally or traditionally have been attacked by compiler technology. Truly mobile code must run on a heterogeneous collection of machines, and therefore must be a machine-independent representation. Liquid software must not corrupt the execution state of the host computer. Liquid software must execute efficiently on the host. Each of these three concerns conflicts with the others. For instance, the host would most efficiently execute binaries of its own machine language, not some machine-independent form that must suffer the overhead of interpretation, or the latency of translation (compilation) to native code. New compiler techniques are therefore needed that minimize the negative effects of these constraints. The goal is to compile liquid software as fast as it can be transmitted over the network—that is, to build a *gigabit compiler*.

For initial evaluation, Java bytecode is being used for the machine-independent code representation [jav95]. Sun Microsystems developed Java bytecode to support mobile code on the World Wide Web. The bytecode achieves machine independence by fixing the byte-ordering, data alignment, calling conventions, data layouts, and so on, to a fixed standard. The bytecode is based on a small stack-oriented instruction set that is easy to interpret and should be simple to translate to native code. Java bytecode was designed so that it can be *statically* checked to determine if it meets a rigid set of security requirements.

The current method of enabling heterogeneous hosts is to have each run a Java bytecode interpreter and verifier. After code arrives at the host, it is verified once and subsequently interpreted as many times as needed. To add a new type of host architecture, one need only port the verifier, interpreter and API. If the host's verifier, interpreter and API are trustworthy, the mobile code can be executed safely on the new host. Creating a portable verifier is not difficult. Neither is porting such an API. Unfortunately, interpreters always add a level of indirection, which must add overhead. To get the machine's native speed, you must directly execute its native code.

To get maximum possible performance, the Java bytecode must be translated on the host to the host's native machine language. To avoid a severe latency problem—this compilation is on the critical path between receiving and executing the mobile code—translation must be done very quickly. Fast code generation techniques, which can generate millions of instructions per second on today's 200MHz processors, will allow us to keep pace with Java bytecode arriving at network rates approaching 1Gbps on the next generation of processors. We will compile the bytecode as it arrives—in the time it takes it to arrive—for immediate execution.

Ultra-fast compilation requires that all compiler components be designed for raw speed. Every compilation phase—from reading bytecodes to emitting native code—must be as fast as possible. Two approaches to developing fast compilers seem necessary: (1) automated tools that generate highly optimized compiler components from concise specifications, and (2) modifications to the Java bytecode distribution format to decrease the overhead of processing the bytecodes.

2.1 Retargetable Compiler Issues

If the host receives machine-independent liquid software, but must execute at native speeds, the mobile code must be translated dynamically to native code [EP94]. The problem of dynamic translation of Java bytecode to native code is complicated by the need to have a (necessarily machine-dependent) translator for every host architecture.

Quickly generating machine code requires three fast compiler components: fast instruction selection, fast instruction assembly, and fast address relocation. Instruction selection is the process of determining which machine instructions will correctly (and efficiently) evaluate the computation expressed by the bytecode. Instruction assembly is the bit-twiddling required to construct machine code instructions from their various field encodings (e.g., opcodes, registers, immediate values, addresses, etc.). Address relocation is the back-patching of address fields in previously emitted instructions when the actual physical locations become known. All of these components are machine dependent. With special attention to details, all can be done efficiently.

Instruction Selection: Efficient instruction selection presents a tradeoff between the importance of generating instructions quickly and generating efficient instruction sequences. It is expensive to generate *very* efficient instruction sequences (for particular bytecodes), but the resulting code would execute quickly. On the other hand, simple, but fast, techniques could quickly generate instructions that would yield slightly slower execution. Optimizing code quickly requires making tradeoffs of analysis time against anticipated efficiency gains—a tradeoff that historically has been made in favor of efficiency gains, which may not make sense when analysis time is indistinguishable from run-time.

Fortunately, retargetable code-generator generator systems can help here [Pro95a]. Code-generator generators can produce optimized code generators for various target architectures from concise machine specifications. We plan to build code-generator generators that produce the code generators that occupy different points along the compile-time/execution-time spectrum—from ultra-fast compile times with modest execution times to fast compile times with excellent execution times.

Liquid software presents some challenges that we are attacking: how to select and emit native instructions quickly, how to optimize code quickly, how to incrementally compile *on-demand*, and how to produce code reliably. Producing native instructions quickly requires a fast mechanism for selecting, assembling, relocating and emitting them—problems that have only recently been approached. Producing code incrementally will greatly increase perceived program speed—if program chunks are compiled on-demand, no time is wasted on unexecuted code.

Instruction Assembly and Relocation: Instruction assembly is inherently machine-dependent. To assemble machine instructions, the appropriate encodings of the various instruction fields must be stitched together to create the desired instruction. This can be straightforward on RISC machines that have notably few instruction formats. For RISC machines, assembling instructions amounts to a sequence of shifts, masks, and OR's at fixed offsets. For the complicated instruction formats that characterize most CISC machines, instruction encodings are far more complicated. The most difficult aspect of CISC encodings is the fact that instructions have many different formats with operands of varying shapes and sizes.

Fortunately, only a finite number of encodings exist for any instruction set. To ensure that our compiler can encode instructions quickly, we are creating a tool that generates optimized instruction assemblers from concise specifications of instruction set formats. (Non-optimizing tools with the same functionality exist [RF95], but they make no effort to optimize the instruction assembler.) These specifications are being coordinated with the specifications used for instruction selection to enable the instruction selector to directly assemble and emit instructions without any delays from intermediate representations.

It may be necessary to assemble instructions before all their fields have been have been instantiated. For instance, a `jump` instruction may need to be built and put into the instruction sequence before the address of its target is known. Once the target is known, the instruction's field must be updated with the correct value. The instruction-assembler generator produces the necessary routines to efficiently patch previously assembled instructions with such late-binding values.

Just-In-Time Compilation: One obvious way to increase *perceived* compilation speeds and hence decrease latency is to defer compilation until code sequences are actually needed. One technique for deferring compilation would be to initially compile each routine into a trivial, short stub routine that would compile that routine when it is first called. This would spread the cost of compilation out over the execution of the program. A similar technique could be utilized at the basic-block level.

2.2 Fast Interpretation

This focus on fast compilation should not be taken to imply that interpreters for liquid software are not interesting. They are extremely important for a variety of reasons: interpreters enable quick porting to new architectures, interpreters can readily provide diagnostic/debugging support that translation systems cannot, and interpreters are easier to verify. We intend to also pursue interpreter-related research, focusing our efforts on the speed and portability of interpreters for liquid software. Interpretation often results in slowdowns of 10-1000$\times$ relative to native execution speeds.

Our research indicates that optimizing code-generation techniques can be applied to interpreter generation to consistently get slowdowns of less than $10\times$ [Pro95b]. These efficient interpreters rely on techniques normally applied to compilers. Techniques include the efficient register use, the efficient instruction schedules, and efficient instruction selection. Other efficiency enhancing techniques are more specifically applicable to interpreters. Caching the top elements of the interpreter's evaluation stack in registers can yield impressive speed gains. Building an interpreter that caches more than just its top value is difficult, however. For this we anticipate building interpreter generators that automatically build highly-optimized interpreters from target machine specifications.

2.3 Verification

Java bytecodes are *verified* before execution. The verification encompasses a variety of static checks that ensure that many safety guarantees of the Java language cannot possibly be violated during execution. The most familiar safety guarantees are those that enforce strict typechecking on the language. For instance, the verifier rejects any program that might try to forge an object reference from an integer or from a differently typed object reference. These guarantees, combined with policies enforced by the API, eliminate the chance of a Java program corrupting the state of the network appliance. Java programs from untrusted sources must be verified. Unfortunately, this verification must precede execution and is, therefore, on the critical path between retrieval and subsequent execution. Any decrease in the time it takes to verify an program would translate directly into decreased latency.

2.4 Scanning and Parsing

The Java-bytecode compiler has a tremendous speed advantage over traditional compilers: the bytecode compiler does not have to preprocess, scan, or parse a readable ASCII program representation. The bytecode distribution format was developed to be a compact, efficient representation of program objects, their types, and the methods (i.e., code) that operate on them. This format should be fast to read and process, but special attention is necessary to read the bytecodes quickly enough to allow gigabit translation speeds. For example, the Java distribution format fixes the internal representation of all integers to be big-endian, but it does not guarantee any alignment restrictions for integer values. To read these values quickly, they must be marshalled efficiently. We plan to automatically generate marshalling code optimized for each target machine, to ensure efficient (and accurate) reading of unaligned data. Experiments have shown that general-purpose marshalling routines used in network protocols can increase speeds by $20\times$ [OPM94].

2.5 Symbol Tables

Both the verifier and the bytecode compiler must maintain symbol-table information in order to properly process the bytecode. The current Java-bytecode distribution format contains enough information to efficiently resolve all symbol table references within a given class. Unfortunately, many references must be resolved between classes, and this requires costly hash-table maintenance. For each inter-class name reference, a hash value must be computed, a lookup must be performed, and any collisions must be eliminated via costly name comparisons. Computing hash values is not cheap. String comparisons to eliminate false collisions—distinct names that have the same hash value—are very costly.

A simple extension to the distribution format could eliminate almost all of this unnecessary overhead. By including a precomputed hash value with every name in the symbol table the work of computing the hash function would be effectively moved from the bytecode compiler/verifier to the originator of the program. Furthermore, making the hash function large (e.g., 32 bits) would virtually eliminate the possibility of a false collision. To keep the hash table size down it could be indexed by the low-order bits of the pre-computed hash value, but the initial comparisons of the name with collisions could be done very cheaply with the full hash value. Only when the full hash values agree does the actual character-by-character comparison need to be done.

3 Mobile Search Applications

The second thrust of the liquid software project is to develop mobile search as an application of liquid software. Currently, searching for information on a network is performed almost exclusively through a client/server environment, where the client supplies the query and the server supplies the user interface, the search engine, and the data. All three of these features can be implemented as liquid software. The following discussion concentrates on the mobility of the search engine; we briefly consider the user interface and the data at the end of this section. Having the search engine originate from the user and run as liquid software on the nodes in the network opens the door to many advantages. We plan to explore these advantages and use search as a testbed for this technology.

Servers can support search without providing any software facilities other than agreeing to run liquid software. When a user initiates a search from a certain document, the user's own search code is transferred to the server, compiled (which is why fast compilation is essential), and then executed. The input to the search could be, for example, the current page plus all the pages pointed to from the current page. Or it could be all pages that can be obtained by following no more than two links, excluding links outside that site and links that the user has already seen (given by the user's recent history file). Or it could be all documents at that site that were updated after a certain date. Or it could be the inverted index supplied by the site. These are just four examples of options specified by the user through the user's own search program. Notice that options can take advantage of quite a bit of information about the user, such as actions already taken (e.g., sites seen), preferences (e.g., case sensitivity), and scope of search (e.g., only new .html files). The user gains much more control over the search process, saves time for adjusting to new interfaces or new search engines, and loses no control over the browsing process. The server provides only the data and the cycles, plus the ability to restrict the search by limiting the amount of time, the number of files opened, and so on. The following summarizes the main advantages of using mobile search code.

Familiarity: Getting used to a search engine to the point of being able to perform effective powerful searches takes time and experience. The current situation on the web where almost each search facility uses a different search engine with different interfaces makes this much harder for users. Allowing users to employ their own search can reduce the learning curve significantly.

Customization: If the search originates from the user, it can be customized much better to that user. Information such as "user profile," "user history" (so that only new results are asked for), and "user preferences" can be sent along with the search. Options such as case sensitivity or size of the output can also be sent automatically.

Post Processing of Results: Some of the search steps can be moved and performed at the user site, reducing the server load. The most obvious one is the output of results. Currently, search facilities spend considerable time on the formatting or *presentation* of answers. For example, putting the right words in bold font, adding HTML formatting, or inserting anchors to lead to the exact place of the match. Furthermore, users often need the answers in a different format and they repeat some of the work. They need to issue a "save as" command to store such answers, and then they may need to convert the answers to a different format (e.g., to include it in another document). With liquid software, the answers can be sent in raw format, formatted on the fly, and merged into the right place by the client machine according to the user's specifications. Furthermore, results of searches from different places using different access methods can be merged together. Not only does this save server resources, but it also makes it possible to customize the answers.

Using Different Search Engines and Simpler Servers: The same information can be searched by different search engines, depending on the users' needs (e.g., flat text search or complex database analysis). By supplying the search engine, users can utilize the same data in many ways. This also relieves the servers from the duty of providing multiple services to multiple needs. One experience we have of running a search server is that we receive too many requests for too many options. With mobile search code, servers can become very simple. They can contain only the raw data in an acceptable format, leaving the task of defining the search to the users. This will also enormously simplify the task of maintaining the data.

Combining and Chaining Results: Users can tie search results from several sources more seamlessly, because they control how the search is done and what is to be done with the results. Furthermore, results of one search can be used as part of another search. We foresee mobile search codes travelling from place to place, collecting and analyzing data, and updating their tasks. The current approach to searching more than one place is to do it in parallel, which is very easy on the web. But this approach does not scale, and if everyone sends their queries to all search facilities at the same time, we would wasting too many resources. A more polite approach would be to send the queries in a chain, with the most likely place first, stopping the process when enough results are collected. This is not only more efficient in terms of resources, it is also much easier to stop (by sending a kill request to places in the middle). When someone by mistake sends a parallel query for a common word to hundreds of servers, all of them may be tied up. We plan to design search algorithms that can travel from place to place and adapt their requirements depending on partial results.

Rapid Prototyping and Adapting to Change: Once a server-based search facility is established it is difficult to change any of its features. If the search comes from the users, it will be much easier for them to incorporate new algorithms or new features, because such changes do not affect anyone else.

3.1 Browsing and Searching

Another aspect of mobile search that we plan to experiment with is the combination of browsing and searching. Browsing and searching are the two main paradigms for finding information on line. The search paradigm has a long history; search facilities of different kinds are available in all computing environments. The browsing paradigm is newer and less ubiquitous, but it is gaining enormous (and unexpected) popularity through the World-Wide Web. Both paradigms have their limitations. Composing search queries that produce the desired results is difficult. Search is also often seen by users as an intimidating "black box" whose content is hidden and whose actions are mysterious. Browsing can make the content come alive, and it is therefore more satisfying to users who get positive reinforcement as they proceed. However, browsing is slow, consumes a lot of compute resources, and users tend to get disoriented and lose their trains of thought and their original goals.

These two paradigms are used separately by most systems. We argue that what is really needed is a combination of the two in a fully dynamic and customizable way. We envision a system where both paradigms are offered *all the time*. In such a system, one would be able to browse freely—the usual hypertext model—and then be able to search from any point. The search will cover only material related to the current document. How to define "related to" is a difficult question. Ideally, one would want to analyze all documents and determine which ones are related, but there could be many different relationships and any such semantic analysis is always very difficult. Running concept similarity filtering on-the-fly may take too much time. In addition, different users may want to define "related to" differently based on needs.

We are currently experimenting with combinations of browsing and searching through our GlimpseHTTP (http://glimpse.cs.arizona.edu/ghttp) and WebGlimpse (http://glimpse.cs.arizona.edu/webglimpse) packages, both of which concentrate on the server side. We plan to provide mobile search code that explores similar ideas from the user side. This will allow us to support several different notions of relationships, different user interfaces, and different a comparison between the two approaches.

3.2 Preprocessing

One of the important issues that large-scale deployment of liquid software will face is how to deal with pre-processing steps. Many applications perform pre-processing to save time or other resources later on. For example, search engines may build indexes to facilitate faster search. A search program that arrives as liquid software will not be able to take advantage of such indexes unless the code is compatible with them. Providing facilities to address such compatibility issues, and in general, to facilitate interaction between liquid software and existing pre-processed information is a major part of this research. Even if there is no one standard for search indexes (and the chances of such a standard are not high at the moment), the mobile search code can understand several indexes, at least to the point of being able to provide minimum facilities, and negotiate the search process with the server. Some servers may opt to provide only one central index for the whole site, and the search code can query that index limiting the search to the relevant URLs. For example, Glimpse and other engines already have facilities to limit search based on file names and switch on-the-fly between indexed and non-indexed search.

3.3 Software Agents

The whole notion of server versus client can be reversed with liquid software: the "servers" may contain some specific information (e.g., employees reports), and the "client" can be a powerful analysis software that gathers all the reports, analyzes them (on the "server" machines), and deposits the results somewhere. In the most general scenario, this client is a software *agent*, completely independent of the servers. Software agents have become quite popular recently, because of many of the same reasons liquid software seems so attractive.

A particular interesting application is one we call *Information Matching*. Think of the problem of finding relevant information as a matching problem between a large set of people and an even larger set of information items. Right now, all searching applications perform the matching by keywords. (That is not to say that they use only Boolean queries, but that the input is only keywords.) Suppose that we construct a set of user preferences, and ask people to evaluate the information they see based on their preferneces. Then, before we match information to people, we match people to people. Given the evaluations and preferences, we can determine that a group of people see things similarly (at least in some limited domain). We can then use the evaluations of some members of the group to suggest that information to other members. If the group is large enough, only a few members will have to evaluate each piece for many more to benefit. This process can be dynamic and self-adjusting. Groups will be changed and reformed; preferences will be adjusted; weights assigned to different attributes will evolve; and so on.

There are obviously many problems that need to be solved to make information matching possible. Two of the most important ones are security and scalability. People will not participate if their privacy is not be ensured, and a central control of the process will not scale. Liquid software has the potential to address both problems.

Our effort concentrates on specific applications—those related to search and retrieval—but in general, our goal is to build the right infrastructure so that agents can be implemented efficiently. We are using the search applications as a vehicle to perform large-scale experiments with liquid software, and at the same time provide very useful tools for others. This follows naturally our work on Glimpse [MW94] and Harvest [BDH+95].

4 System Infrastructure

We are building a complete system to demonstrate liquid software. The system supports the mobile search application implemented in Java and a fast and dynamic compiler for Java bytecode, all running on top of Scout—a communication-oriented OS explicitly designed to support customization [MMO+95]. Once completed, this platform will be used to experiment with liquid software as an approach to building large-scale, dynamically-customizable networks.

Our main objective in pursuing this demonstration system is to gain experience with liquid software. While we expect the unexpected, we do foresee two specific issues that must be addressed: (1) the definition of the interface between the framework that supports mobile code (e.g., Java) and the base system (i.e., the underlying OS), and (2) how the OS on each node protects its resources from erroneous and malicious programs. These issues are crucial to the successful deployment of liquid software, and they are the part of the system that is least understood. The experience we gain implementing and using the demonstration system will add significantly to our understanding of these issues. Consider each issue, in turn.

4.1 Liquid Software API

In contrast to traditional Remote Procedure Calls (RPC) systems, in which the interface between clients and servers is static and fixed, liquid software promises greater flexibility by allowing the application itself to define the interface between the client and the server. One way to view liquid software is as a dynamically configurable RPC system. The client defines the interface and semantics of the RPCs by downloading the appropriate code onto the server. Thus the application can tailor the client/server interface to the task at hand, avoiding the need for a single, complicated, and inflexible RPC interface.

While much of the current rhetoric surrounding systems that support mobile code has focused on their ability to provide flexible client/server interfaces, there is a danger in assuming that such a system automatically solves the inflexible interface problem. In reality it does not, and in fact, it may exacerbate the situation by pushing the problem down to the lower levels of the system where it is not as readily apparent. At some level the system must provide interfaces that are static, on top of which the dynamic interfaces are constructed. For example, the infrastructure for liquid software must provide an interface that is used by the applications to install software on a remote machine. This interface sits "below" the liquid software, and therefore cannot be extended or modified using liquid software.

The danger in designing a liquid software system is in failing to recognize that not only do static interfaces exist, but that they are also crucial to the success of the system. If the static interfaces are not designed correctly they may unintentionally limit the functionality and performance of the software that uses them. Micro-kernel operating systems offer a concrete example of this peril. Touted as a mechanism for implementing flexible operating system services, in reality they created lower-level static interfaces of their own that often proved inadequate [Kup93].

The static interface of primary concern for liquid software is that of the application programmer's interface, or API. The API is the interface to the services provided by the underlying operating system. Since liquid software is not used to implement the underlying system the API is necessarily static, and hence may limit the functionality of the liquid software if not designed properly. Thus, when designing the API, care must be taken to ensure that it is complete, minimal, portable and properly addresses security concerns. The API should be complete so that it does not prove to be a limitation in the future, yet it should be minimal lest it become cumbersome to use.

Consider an application that needs to know the last time a file was modified, or the type of graphics device installed in the machine. If the API does not provide this information, there is little the application can do to rectify the problem. On the other hand, adding every conceivable routine to the API is not a reasonable option either. There is certainly no way to design an API and prove that it is both complete and minimal; the best we can hope for in this area is to experiment with liquid software enough to gain insight into what the API should contain. Many of the good ideas in operating system design should also carry over. The API problem is made even worse, however, in that defaulting to the "least common denominator" often does not suffice. UNIX, for example, does not enforce a format on a file it stores, simply treating it as a string of bytes. This does not mean, however, that such a file does not contain an internal structure that must be understood by the applications that use it. A prime example of this problem is the index files used to improve search performance, as described in the previous section. The internal structure of these index files defines an interface *outside of the system* to which the software must adhere in order to interact.

The liquid software API must also be portable, as well as complete and minimal. Liquid software is intended to run on a heterogeneous collection of machines, encompassing many different architectures and operating systems. All of the functions provided by the API must therefore be implementable on each of these different platforms. Certainly there may be platforms that are too limited to support the full API (e.g., a network video camera may only support a subset), but the full API should work on all current and future standard computing platforms. Care must be taken so that current operating system idiosyncrasy, such as limits on file name lengths, number of processes, etc. are not hard-wired into the API such that they eventually cripple it.

Finally, the liquid software API must address security. As described in the following subsection, there is a synergy between verification and trust techniques and the API. Security is provided through their interaction, and shifting the security functions among them can have dramatic effects on performance and flexibility. For example, one way to provide security within the file system is to disallow liquid software from using files by not having any file access routines in the API. This solution is high performance, but is inflexible and perhaps overly constraining. At the other extreme protection can be provided through verification and trust, which is highly flexible but requires verification checks to be performed before running the software. The most desirable solution is for the API to provide the underlying security mechanisms, while the verification layer implements the policies that use the mechanisms. There are many considerations that must be addressed, however, including what are the proper abstractions for the API to provide (e.g., does file protection based on user ID make sense for liquid software?) and the need for the API to be portable (i.e. the API's security mechanisms must be implementable using those provided by the underlying OS). As for the completeness and minimal concerns the answers to these questions can only be had through experimentation.

From all of these considerations, we conclude that liquid software is not a magic bullet—it has simply changed the nature of the interface design problem, not eliminated it altogether. Liquid software may allow flexibility at the client/server interface, but it only pushes the problem down to the underlying inflexible API. Therefore, a major focus our effort is to study the API requirements of liquid software and design an appropriate interface. A demonstration system consisting of a mobile searching application implemented in Java and running on top of Scout gives us a realistic environment to conduct this study.

4.2 Safe Liquid Software

The ability of liquid software to readily flow from computer to computer is both a blessing and a curse. It is a blessing because enables a new computing paradigm—mobile code that dynamically extends the functionality of the computers it visits. On the other hand, the very mobility of liquid software is a curse because it opens a huge potential security hole. Traditionally the outside world can only interact with a computer through well-defined interfaces presented by the fixed set of programs installed on the computer. These restrictions allow the computer to protect itself from external attack. Liquid software eliminates this protection, however, since the code that the computer runs is provided by the very external agents from whom the computer should be protected. Without the proper precautions the computer may import and run unsafe code, thereby opening up the computer to abuse through misuse of its resources, e.g. the code may consume excessive amounts of memory or CPU, access memory or files which it should not, leak information to the outside world, and so on.

The solutions to the problem of unsafe liquid software fall into three categories: user control, implicit trust, and verified access. User control is the simplest of the three to implement, and is the one currently used in mobile code systems such as Java. In this solution the user controls the resources that the liquid software may access, either by setting up access control lists for the resources, or by responding to dialog boxes that describe the type of resource that the liquid software requires and allow the user to decide whether or not to grant the request. For example, the user may use access control lists to specify that the default liquid software may not access any files, or perhaps only files in a temporary directory. Should liquid software attempt to do so it is either terminated, or a dialog box is presented to the user indicating which file the software is trying to access and asking the user if it should be allowed to do so.

User control of resources is simple to implement because it is only a mechanism and leaves all policy decisions to the user. There are several problems with this approach, however. First, it may require excessive user interaction to control the resources available to the liquid software. If the user is presented with too many dialog boxes and must make too many changes to the access control lists there is a danger that either the user will decide not to use the liquid software in question (thus defeating its purpose) or the user may simply circumvent the security system by allowing access to all resources. Furthermore, it is our assertion that any non-trivial piece of liquid software will need to access resources in a potentially dangerous fashion. Access to user files is probably the best example of this. What makes user control difficult is that for the users to make intelligent decisions about which liquid software should be able to access which files they will need an in-depth understanding of what the software is trying to do and why it needs access to the files. This implies a high level of technical sophistication on the part of the users, something that we expect the average consumer of liquid software will not have. Thus user control is not a viable option for making liquid software safe; it requires high levels of sophistication and interaction on the part of the users.

At the other extreme from user control is that of implicit trust. In this solution liquid software is considered safe once it is confirmed that the software originated at a trusted entity. Consider a piece of liquid software that computes the value of one's stock portfolio. To do this it must not only have access the files that describe the portfolio, but it also must have access to the network to obtain the latest stock prices. There is no practical way of ensuring that such software will not abuse the resources it needs to do its job. Once it has access to the files and the network it could leak one's portfolio to the world, for example. Clearly it is infeasible for the user to carefully examine the source code to check its correctness, nor will automatic methods work since this problem is equivalent to the halting problem. The only feasible way of running this software is to implicitly trust it, and therefore the person or entity that created it.

While it may seem rash to simply trust software based upon its origin, this is exactly what is done today with shrink-wrapped software. The name on the box tells the consumer who produced the software and therefore is to blame should there be a problem. With this reassurance consumers regularly install software on their systems without any restrictions on the resources that the software may access, leaving them totally vulnerable to improper behavior by the software but confident that they at least know who to hold responsible.

What is needed to obtain this same level of trust in liquid software is the electronic version of shrink-wrap, or in other words, digital signatures. Liquid software is wrapped in *electronic shrink-wrap* by having its creator digitally sign it. When a consumer receives a piece of liquid software this electronic shrink-wrap allows them to make an informed decision about the origin of the software and whether or not they trust that entity enough to run the software. Thus the problem of user control over resource access boils down to the user deciding which entities he or she is willing to trust.

Although electronic shrink-wrap allows the liquid software consumer to verify its producer, this does not mean that it is wise for the consumer to allow the software unrestricted access to the computer's resources. At the very least there may be bugs in the software that cause it to behave in unexpected ways. Because of this it is desirable to run a variety of safety checks on the liquid software to verify its integrity prior to running it, even though its producer is trusted. This verification spans a spectrum from very low-level checks such as those that must be done on Java bytecode to ensure that it does not violate the safety guarantees made by the language, to very high-level checks such as those necessary to ensure that the software does not access inappropriate files. In general, the higher the level of the checks the more complex safety concerns they address. Low-level checks may be sufficient for catching coding errors, but high-level checks are needed to catch malicious attempts to violate security. It is important to remember that verification cannot completely replace trust since verification in general is equivalent to the halting problem. Thus truly malicious attempts to violate security can only be thwarted via electronic shrink-wrap.

One of the interesting possibilities that fall out of the relationship between trust and verification is that it is possible to do the verification process prior to load-time. To do this a third entity is created whom the consumer trusts, and this entity verifies the software and adds its own electronic shrink-wrap to signify to the consumer that the software has been verified. With this assurance in hand the liquid software can be run without re-verification. Once again, while this approach may seem radical it has a parallel today in the Underwriter's Laboratory (UL). The UL produces safety guidelines that manufactured products must follow, thus eliminating the need for the consumer to perform the checks himself. For example, the UL sticker on a toaster informs the customer that the internal wiring and components meet basic safety guidelines, therefore the customer does not need to take the toaster apart to ensure that it will not cause an electric shock or fire. (Note, however, that the UL label does not say anything about the toaster's ability to make toast). Through this process the consumer is able to replace the need to personally verify the safety of products with trust in UL. In a similar fashion we plan to develop a Liquid Software Underwriter's Laboratory whose purpose is to perform a set of basic verifications on software and electronically shrink-wrap the result. This reduces the amount of verification that must be done to run the software to simply validating the integrity of the shrink-wrap.

The liquid software infrastructure we are developing both provides the mechanisms necessary to support implicit trust and verification, as well as serves as a test-bed for investigating the trade-offs between the two. Our work with electronic shrink-wrap will primarily be on the infrastructure aspect; we assume that others have adequately addressed the cryptographic issues associated with digital signatures and we plan incorporate their work into the infrastructure (we are currently planning on using the PGP package). We are instead turning our attention to the mechanism for distributing the keys for the shrink-wrap, and the user interface to the the entire shrink-wrap mechanism. By building a working liquid software system we hope to experiment with the utility of electronic shrink-wrap in general.

On the verification front we are experimenting with the utility of various degrees of verification, the effect that verification has on the liquid software API, and the benefit of having a centralized Underwriter's Laboratory for liquid software. The gigabit compiler effort needs to address the low-level verification required to support Java. We are extending this verification to the API itself and experiment with the usefulness of verifying different aspects of how the software uses the API. For example, one possible verification is that the liquid software only modifies files which it itself created. Once a piece of software passes this test the user can be assured that running it will not modify any existing data. If the software is question is so complex that the verifier cannot make a determination about its safety the user will be notified so that he or she can decide whether or not to run it.

We also intend to experiment with the liquid software API and its effect on the verification process. For example, one could imagine an API that restricted the software to only modifying files that it created. This would eliminate the need for the verification mentioned above, at the cost of limiting the functionality of the software. Nonetheless, these tradeoffs need to be examined and experimented with, something the liquid software infrastructure will allow us to do.

5 Related Work

We divide the research related to liquid software into three areas corresponding to the three major themes of our effort: compiler techniques, mobile search, and system infrastructure.

5.1 Compiler Techniques

Dynamic code generation—the generation of code at run time for subsequent execution—is currently a popular research area. Dynamic code generation is routinely studied as it relates to generating highly specialized routines optimized with respect to values that could only be known at run time. Leone and Lee's system automatically generates specialized routines for a functional language, Fabius [LL94]. Their system does not expose any details of the code generation to the user, and operates relatively efficiently.

Engler and Proebsting took a more nuts-and-bolts approach to dynamic code generation with their DCG system [EP94]. At run time users specify code to be emitted in a machine-independent tree format which is subsequently translated by DCG to binary code for immediate execution. The translation is efficient, executing about 300 instructions for every instruction generated. The DCG system is constructed by tools that translate machine specifications into the necessary routines for dynamic code generation. The machine specifications indicate the translation from the tree format to machine instructions, and the binary format of those instructions. Porting DCG to a new platform requires creating new machine specifications.

Engler has created a newer, more efficient dynamic code generation system called VCODE [Eng96]. VCODE is a machine-independent intermediate representation that resembles 3-address code for a RISC-like instruction set architecture. The VCODE is simple to translate (suboptimally) to machine code. The translation system requires about 20 instructions per instruction generated. Like DCG the VCODE system is built with tools that process machine specifications. VCODE is currently the target of a high-level language, ‘C, that is designed to ease the development of programs that rely on dynamic code generation [PEK96].

Ramsey and Fernandez have developed the New Jersey Machine Code Toolkit, which translates a specification of a machine's instruction set into C routines that can assemble and disassemble binary images of those instructions. The generated routines can also handle the relocation of addresses within previously assembled instructions. Unfortunately, the generated routines, while useful, are not optimized for speed.

Franz developed a system for generating code "on the fly" for Oberon programs [Fra94]. Like Java, Oberon modules load dynamically, piece by piece. Franz's system loaded compiled modules as abstract syntax trees for immediate translation to machine code. Because the system read modules from (slow) disks, the trees were much more compact than the resulting object code, and his code generator was very fast, it was actually faster to read *and compile* the trees than to simply read object code directly.

Many people have worked on making interpreters fast. Much of the work has centered around the time/space tradeoffs for various encodings of the interpreted code (e.g., direct-threaded vs. indirect threaded)[Pit87, Kli81]. Ertl demonstrated the speed advantages and complexities involved in caching multiple stack elements in registers in a stack-based interpreter [Ert95].

5.2 Mobile Search

The closest area to mobile search is the area of intelligent information *agents*, which have received a lot of deserved attention lately [CAC94, WJ95]. Attempts to combine several search engines has been carried out most prominently by the *meta-crawler* [SE95]. Wiederhold [Wie92] (and many others) suggest *mediators*, which mediate between the users, their data, and their application software. These mediators are assumed to reside in the servers, but they could also be implemented as liquid software.

Using user preferences and histories has been incorporated into several search applications including NetAgent [PC95], which allows users to share common views and feedback through collaborative indexing (they used our Glimpse system for their indexing), INFOSCOPE [FS91], which makes suggestions based on previous usage patterns, and the *Expertise Locator* [KMS95] of Bell Labs for locating experts in any particular topic (they also use our Glimpse system). Our work with Glimpse is described in the next section.

5.3 System Infrastructure

While the machine-independent bytecode aspects of liquid software has received considerable attention recently, comparitively little effort has been put into the system-related issues. There is little experience—beyond simple "toy" demonstrations—with the Java API, and the Java protection mechanisms are crude at best (security is either turned on or off). A major thrust of this effort is to implement a realistic application using liquid software so we can gain experience with these issues.

OS support for liquid software is also a mostly unexplored area. The most commonly employed strategy, as exemplified by Sun Microsystems, is to pare back a full-fledged operating system (e.g., Solaris), and use it as a Java kernel. This is a short-term solution, at best, considering the unweildy nature of modern day operating systems. In contrast, Scout will allow us to configure precisely the modules required by the Java API. In other words, one can configure a Scout kernel to support exactly the Java API.

References

[BDH+95] C. M. Bowman, P. B. Danzig, D. R. Hardy, U. Manber, and M. F. Schwartz. The Harvest information discovery and access system. *Computer Networks and ISDN Systems*, 28:119–125, 1995.

[CAC94] Special issue on intelligent agents. *Communications of the ACM*, 37, July 1994.

[Eng96] D. R. Engler. VCODE: a retargetable, extensible, very fast dynamic code generation system. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, May 1996. To appear.

[EP94] Dawson R. Engler and Todd A. Proebsting. DCG: An efficient, retargetable dynamic code generation system. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 263–273, October 1994.

[Ert95] M. Anton Ertl. Stack caching for interpreters. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 315–327, June 1995.

[Fra94] Michael Steffan Oliver Franz. *Code-Generation On-the-Fly: A Key to Portable Software*. PhD thesis, Swiss Federal Institute of Technology Zurich, 1994.

[FS91] G. Fischer and C. Stevens. Information access in complex, poorly structured information spaces. *Human Factors in Comp. Sys. (CHI91) proc*, pages 63–70, 1991.

[jav95] The JAVA language: A white paper. Technical report, Sun Microsystems, 1995.

[Kli81] Paul Klint. Interpretation techniques. *Software Practice and Experience*, 11(10):963–973, October 1981.

[KMS95] H. Kautz, A. Milewski, and B. Selman. Agent amplified communication. *AAAI Spring Symposium Series on Information Gathering from Distributed, Heterogeneous Environments*, 1995.

[Kup93] Michael D. Kupfer. Sprite on Mach. *Proceedings of the Third USENIX Mach Symposium*, pages 307–322, April 1993.

[LL94] Mark Leone and Peter Lee. Lightweight run-time code generation. In *Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pages 97–106, June 1994.

[MMO+95] A. Montz, D. Mosberger, S. W. O'Malley, L. Peterson, and T. Proebsting. Scout: A communications-oriented operating system. In *Proceedings of the Fifth HotOS Workshop*, May 1995.

[MW94] U. Manber and S. Wu. GLIMPSE: A tool to search through entire file systems. In *Usenix Winter 1994 Technical Conference*, pages 23–32, January 1994.

[OPM94] Sean O'Malley, Todd A. Proebsting, and A. Brady Montz. USC: A universal stub compiler. In *Proceedings of SIGCOMM 94 Conference on Communications Architectures, Protocols and Applications*, pages 295–306, August 1994.

[PC95] Taeha Park and Kilnam Chon. Netagent: A global search system over internet resources by distributed agents. In *INET'95 Hypermedia Proceedings*, 1995.

[PEK96] Massimiliano Poletto, Dawson R. Engler, and M. Frans Kaashoek. tcc: A template-based compiler for 'C. In *Proceedings of the Workshop on Compiler Support for Systems Software*, pages 1–7, February 1996.

[Pit87] T. Pittman. Two-level hybrid interpreter/native code execution for combined space-time program efficiency. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 150–152, June 1987.

[Pro95a] Todd A. Proebsting. BURS automata generation. *ACM Transactions on Programming Languages and Systems*, 17(3):461–486, May 1995.

[Pro95b] Todd A. Proebsting. Optimizing an ANSI C interpreter with superoperators. In *Annual Symposium on Principles of Programming Languages*, pages 322–332, January 1995.

[RF95] Norman Ramsey and Mary F. Fernandez. The New Jersey machine-code toolkit. In *Proceedings of the 1995 Usenix Winter Conference*, January 1995.

[SE95] Erik Selberg and Oren Etzioni. Multi-service search and comparison using the metacrawler. In *Proceedings of the 4th World Wide Web Conference*, pages 195–208, 1995.

[Wie92] G. Wiederhold. Mediators in the architecture of future information systems. *IEEE Computer*, pages 38–49, March 1992.

[WJ95] M. Wooldridge and N. R. Jennings, editors. *Intelligent Agents - Theories, Architectures, and Languages*, volume 890. Springer-Verlag, 1995.