The University of Texas at Austin, Department of Computer Sciences. Technical Report CS-TR-01-40. October 2001.

A Memory Accounting Interface for The Java Programming Language*

Mirza Beg, Mike Dahlin
beg@alumni.utexas.net, dahlin@cs.utexas.edu
Department of Computer Sciences
The University of Texas at Austin

ABSTRACT

Widespread use of the Internet infrastructure for deploying services creates new issues and raises serious concerns regarding the security of their execution environment. Ideas of employing dynamic distributed systems for mounting e-services on the web are gaining strength. The main idea behind their proposed design is the use of distributed extensions. This permits execution of un-trusted service code at clients, content distribution service machines or proxies, in order to make the dynamic services more effective.

Over the past few years Java has surfaced as an attractive option for constructing web services and programming their execution environment. Java provides the capability of automatic memory operations but fails to provide an accounting interface. In order to make the services more secure the language needs a robust resource accounting interface.

This paper discusses the design and implementation of a memory accounting interface as a key component of resource management. We discuss the design, implementation and issues regarding the implementation of this system. To consider its practical application, we evaluate the performance and accuracy of this system.

KEYWORDS

Java, Resource Management, Security, Bytecode Rewriting, Mobile Services, Un-trusted Code.

1. INTRODUCTION

This paper examines the advantages and practicality of a memory management interface for Java. The object is to lower the denial of service risk which has escalated due to increasing use of active services over the web, the circulation of un-trusted code from unreliable sources and of possibly aggressive intentions. This activity can pose a potential security threat on both the client side and server side likewise, especially in the presence of mobile service code [13].

Java has surfaced as an appropriate choice for creating extensible services and deploying them on the Internet. This extension demands the Java Runtime Environment to provide adequate security as well as a fair distribution of resources among services. Java applets are a prime example of downloadable content executing on a client. Disallowing access to the disk and network connectivity to this piece of un-trusted code by executing it in a black box environment does protect the user from a potential security breach, not from denial-of-service. For example, there are no limits to the amount of memory this code can use, thus allowing it to utilize memory resources at its discretion. This scenario illustrates a potential threat of denial-of-service by crashing the browser. Simply consuming the available memory would render the JVM unable to function normally. Similarly server extensions can pose an even greater threat to service providers. Java servlets are an example of such extensions. Such uploadable content can potentially disable the server to a state where it is no longer capable of processing any further requests.

In the current implementation of Java (version 1.3) [9] an interface for memory resource management is absent i.e. a system using Java as its execution environment is unable to control resource distribution between services. Unless the Java runtime environment has the ability to associate memory allocations to their allocating codelets (through their respective threads) can limit memory consumption, the task of deploying extensible systems seems impractical.

In this paper we propose a memory management system for Java. This interface accounts for heap memory on a per thread basis. The system is designed to associate all allocated live objects to their respective allocating threads and hence account for the total memory allocated by each thread. In addition the system sets limits on memory consumption for each thread. The system also terminates threads of misbehaving code.

The main contribution of the paper is to motivate the incorporation of a memory management module in the Java Runtime Environment. The goal here is to show that this can be done without modifying the underlying structure of the Java Virtual Machine. Including such an interface in the runtime environment can be done with a

* This work was supported in part the Texas Advanced Technology Program, the Texas Advanced Research Program, and a grant from Novell. Dahlin was also supported by an NSF CAREER award (CCR-9733842) and an Alfred P. Sloan Research Fellowship.

reasonable overhead and this is demonstrated by the performance of the prototype system.

The rest of the paper is structured as follows. In the next section we discuss the motivating factors for our work. Then we describe the architecture of the prototype system, this section is followed by a section on experimental results. In the following sections we discuss the issues with the current implementation, some related and ongoing work, and we conclude by evaluating the practical applications of the system.

2. MOTIVATION

Two main factors have motivated the development of this prototype system. The first being the absence of a resource management interface in the language specification. Second being the need for additional security, which is essential for stable execution environments. We gather most of our motivation from the intended use of Java for extensible environments. In order to be practical, such environments need to control the access to resources and be able to enforce limits on consumption.

A leading example of extensible service deployment would be the Active Names System [2], which allows un-trusted code to execute dynamically over the network with the intention of improving availability of wide-area services. In this system a client can upload service extensions to customize the available services. This uploaded code is then available to a large group of other potential clients employing the same system for accessing web services whether through a simple web browser or a more sophisticated mode. The important thing to note here is that both the system core and the uploaded code are being executed on the same Virtual Machine. They are essentially running in different security domains yet there is no distinction between the two as far as resource access is concerned. Hence the system is susceptible to denial-of-service attacks unless a resource management unit is added to improve and restrict resource distribution and consumption.

The absence of a resource management interface can have serious consequences both for the service provider and the end-user. The un-trusted codelet can exhaust the available memory and cause starvation in all other applications executing on the system. Such an attack can not only cause denial-of-service by not allowing other requests to be processed by the system but can also lead to a system wide crash and prevent new processes from initiating.

From the end-users point of view it would be desirable to have a memory management interface that the system can use to prevent un-trusted code from using excessive memory in order to prevent the denial-of-service attacks on the client side. The potential threat to the end-user comes primarily from web applets and other downloadable Java content. Code with malicious intentions can overwhelm the system by extensively consuming memory and possibly resulting in application failures and even system crashes, in the worst scenario.

Denial-of-service attacks can disable a system through all the resources available to the un-trusted code. To deal with these attacks work is currently being done for the management disk and network resources [3].

3. PRELUDE TO SYSTEM IMPLEMENTATION

The implementation of the memory management interface could be simpler if the code was trusted and the programmers could be trusted to send notifications for memory manipulations. Unfortunately there are a few problems with this approach. Firstly, we cannot trust the programmer. Secondly, even if we did trust the programmer there is always the risk of human error. And lastly, the process is entirely mechanical.

Keeping the above-mentioned limitations in mind, we decided upon an automated module, *bytecode-rewrite*, to rewrite the compiled java `.class` files to insert the notification callbacks to the memory manager. The details of bytecode rewriting are discussed in Section 4.3, explaining the engineering of these modifications.

4. MEMORY MANAGEMENT: THE PROPOSED INTERFACE

Faced with the challenges discussed in Section 2, our goal is to construct a flexible Memory Management Interface without modifying the existing structure of the Java Virtual Machine. Modifying the JVM would mean sacrificing the portability of java code.

Memory accounting in this system is accomplished through bytecode engineering; by adding notifications signals upon state changes in heap memory. This is accomplished by inserting appropriate bytecode instructions at selected places in the original classes in order to maintain information about memory consumption recorded on a per-thread basis.

Our implementation consists of three main components: A *memory accounting* module, that provides an accountability interface for heap memory, a *management policy* module that uses a policy to set limits to memory consumption and enforces them, and a *bytecode-rewriting* module that inserts callbacks in appropriate locations in the code. These modules are individually discussed in the following subsections.

4.1 MEMORY ACCOUNTING

The objective of the memory management interface is to know, at any given time, the amount of memory consumed by each individual thread currently executing in the system.

This interface is designed to receive notifications for any activity in the heap memory of the system. This means that when an object is allocated in the memory space, a notification is sent to the Memory Manager. A similar notification is sent upon object de-allocation. These notifications are received as function calls to the Memory Manager, within which these signals are handled appropriately.

This module is responsible for keeping the current state of memory distribution in the heap. It keeps track of all threads in the system, the references to their live objects and the amount of memory consumed by these objects measured in bytes. When an object is allocated a notification arrives at the memory manager along with the information regarding the allocating thread and the size of the object. At this time the allocating thread is charged for the space occupied by this object. When this object is de-allocated and is collected by the Garbage Collector another notification arrives at the memory manager, signifying the expiration of this object. As a response to this callback the memory manager subtracts the space occupied by this object from the usage account of its allocating thread.

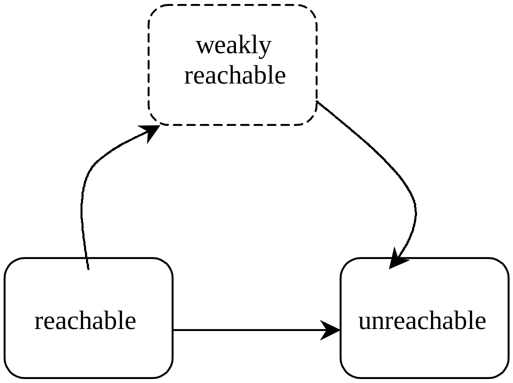

Figure 1: State transitions for reclaiming an object.

Accounting for memory allocated by arrays is a little trickier. When an array allocation notification is received a *weak pointer* for this array object is saved in the account of the allocating thread, in addition to incrementing its usage by the amount of memory allocated by this array. This value is calculated by multiplying the array length to the size of a single array object. In the case of multidimensional arrays the dimensions are multiplied out to calculate the size of the array. In the case of a primitive array the dimension is multiplied by the corresponding primitive size.

When the object is no longer reachable from any part of the code it is garbage collected. Figure 1 shows the state transitions an object goes through during its lifetime. Weak pointers do not obstruct de-allocations of objects. Thus the array-objects, which are garbage collected, their corresponding weak pointers turn to *null*. The sizes of these array-objects are subtracted from the total memory usage of the thread after they have been garbage collected and the memory manager finds their references to be *null*.

4.2 MANAGEMENT POLICY

This module performs three essential functions. (i) Registers new threads (ii) sets an upper limit to the memory that can be consumed by a particular thread (iii) handles overuse callbacks and takes appropriate action whenever memory usage of a particular thread exceeds its limit.

Registration. Thread registration is one of the most important steps in memory accountability. When a thread allocates an object in heap memory for the first time, it is registered with the memory manager. This registration initializes the parameters in the memory manager for the corresponding thread. The purpose of this registration is to keep track of all the threads whether active or not, that have made use of the memory heap. This means that threads that do not perform any dynamic memory allocations are not registered at all. Once the thread is registered, the memory manager can proceed with the memory usage data that becomes available.

Limits. The limits to memory consumption are set at the time of thread registration by this policy module. The limit for a particular thread is a numerical value, which acts as an upper bound on the memory bytes that can be consumed by this thread. The limit can be assigned a fixed value but a more plausible idea would be to assign it a value that is a percentage of the available free memory. Yet a better idea is to let this limiting value be dynamic. In service providing systems, higher limits can be set for more popular services and lower for the least popular ones. The schema of popularity based resource management has been developed on the notion that popular services should be given higher priority. Practical application and prototype implementation of such a system is discussed in the proposed disk-space management system [3]. For the purposes of the experiments described in this paper we used constant limits.

Enforcement. Memory overuse callbacks are currently handled by terminating the thread and freeing the memory resources consumed by it. This module is responsible for accomplishing thread termination, when required. It is important that the threads are terminated softly so that any locks held are released before the thread is killed to avoid a deadlock in the system. Dr. Wallach has done a

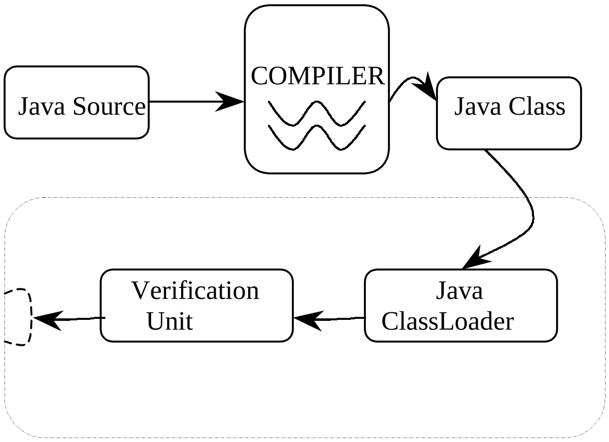

Figure 2. The normal stages a simple java program goes through before it is executed.

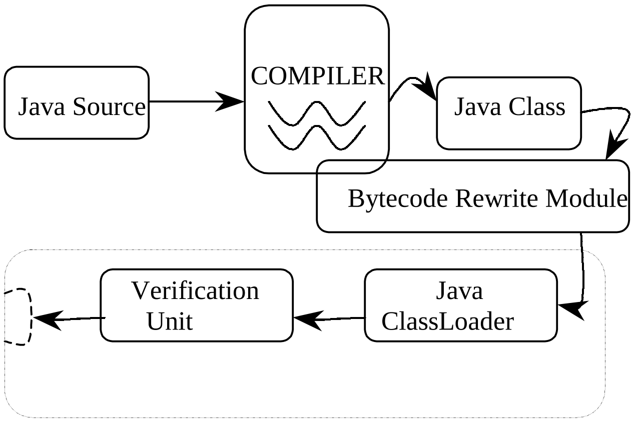

Figure 3. The modification to the pre-runtime stages of a simple program to add hooks into Java bytecode.

significant amount of research in the area of Soft Termination in Java Runtime [1]. In the prototype discussed here we force the thread to throw a ThreadDeathError, which releases the locks held by this thread after which it terminates the thread.

4.3 BYTECODE REWRITE

Bytecode engineering has been employed in this system to implement the notification mechanisms for activity in the memory heap. This means that the notifications received by the memory manager discussed in Section 4.1 are inserted into the code by this bytecode rewriting module. Notifying function calls are inserted in appropriate places in the code to detect changes in the state of the memory heap.

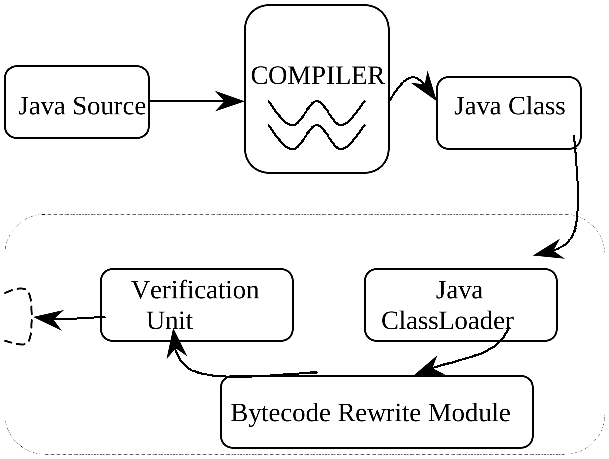

Figure 4. The modifications to runtime stages when bytecodes are written online i.e. at runtime

The rewriting has been accomplished with the help of a bytecode modification toolkit developed at IBM, namely jikesBT [4]. An evaluation version of this toolkit can be obtained from the alphaworks website. The programming interface provided by this toolkit allows the programmer to dwell into java bytecode and experience the flavor of the stack based java environment [11].

The modifications to the Java classes are completed before runtime i.e. the classes are modified after compilation and before execution. These modifications to the class files are discussed in the following paragraphs

.

Allocation Detection. The code for every method is modified to send callbacks to the memory manager upon every allocation of a Java object or array after the allocating instruction. This callback is made in the form of a function call to the memory manager and the required information is passed to the interface as function parameters. As a result of this notification the accountability interface credits this memory allocation to the account of the allocating thread.

De-allocation Detection. Before an object is collected and its space freed, the finalizer function is called by the system garbage collector. We use this feature in Java to notify the memory management system of object de-allocation. We insert a callback to the memory manager at the end of the finalizer code. This piece of code is executed when the garbage collector runs the finalizer. When this notification is received by the accounting interface, the space occupied by the collected object is subtracted from the allocating threads account.

Obtaining Object Size. An additional function (*public static __sizeof()*) is added to each rewritten class to obtain the size of the allocated object. This is calculated as the sum of its fields' sizes. This function is made static so that it can be called to calculate the array sizes.

Allocating Thread. A public field (*__allocator*) is added to the code of each class to record the information regarding the allocating thread. This field is also used as a rewrite flag to prevent classes from being rewritten more than once.

Initialization. Constructors of non-array objects are modified to initialize the *__allocator* field to record the allocating thread identification. Thus when an object is allocated it records its allocating thread as being the current thread. This information is also used when an object is de-allocated to obtain allocator identification. If a constructor is not present, a default constructor is created.

Bytecode modifications can be accomplished before runtime as shown in Figure 2. but some environments may require this rewriting to occur during runtime, as shown in Figure 3. Although both approaches look similar, moving the rewrite module one step ahead in the execution schedule is difficult and greatly affects the performance of the system. The challenging part in bytecode engineering is to maintain the consistency of the stack based execution model in Java. This is crucial for a class to pass through verification while the class is being loaded for execution. Java Stack Inspection [5] provides details of maintaining and securing the stack-based model using a more formal approach.

4.3.1 RESTRICTIONS

The design of the system creates some limitations for the programmer.

The use of finalizers has been restricted in order to seal a backdoor passage into memory. The finalizer has been called on an object after its un-reachability determined by the garbage collector. Once the finalizer is executed, the object cannot be re-incarnated.

The use of variables inserted by the bytecode-rewrite module is restricted. The programmer cannot use variable or function names *(__allocator, __sizeof()).* This restriction is to prevent verification conflicts in the code.

Access to memory manager functions is restricted, so that the user cannot trick the memory manager by sending false notifications.

If the code fails to comply with the above-mentioned rules, it will be restricted from passing the verification phase and as a result would not be loaded into the JVM.

5. DISCUSSION

The memory management system proposed in this paper was designed using the technique of bytecode engineering. This strategy raised certain issues during the implementation of the system. These issues are discussed in greater detail in this section. Also there is this noteworthy relationship between heap allocations and stack consumption, this topic also needs some clarification.

5.1 ISSUES

Although the objective of building a memory management system was to be able to monitor memory activity to its entirety, there are certain forms of memory activity that the current system is unable to detect.

The technique of sending notifications to the accountability interface upon every allocation and de-allocation works fine for memory manipulations in the non-system code. The problem arises when a system object allocates another system object. The system classes are not modified to have callbacks. These files are a part of the standard JVM and modifying them would mean loosening the JVM standards and consequently making this system non-portable.

The system files could be modified while they are being loaded at runtime. This would prevent us form modifying the JVM structure, but system-class-loading in existing versions of the java runtime environment is done exclusively by the system class-loader, preventing us from accounting for system objects allocated by system code.

The important point to note here is that non-system objects allocated by system classes are still accounted for by the memory manager because their code is continuously monitored. In conclusion we found that with the current tools system code is an exception to memory accountability.

5.2 STACK ALLOCATIONS

A significant point to note is that the memory manager described in this system only accounts for the heap memory utilized by the system's threads. It does not account for automatic allocations done on the program stack. These allocations are not a concern here because each Java thread has its own stack space initially set at a default size of 2 megabytes. The limits are enforced by the system. Program threads that overflow the allocated stack will receive java.lang.StackOverFlowException. Since the limitations on stack space are enforced by the system itself and individual thread stacks cannot interfere with each other, the stack space is not a potential target for attacks via denial-of-service.

Dynamic allocations create objects on the heap. This memory heap is common to all the threads running in the process. As a consequence a single thread can consume most of the heap memory and cause starvation. As a result other threads running in the system may not be able to perform their functions and in severe cases the system may not be able to process any further requests. Individual thread stacks cannot pose such a threat to the system simply because they cannot occupy the stack space reserved for other threads in the system.

If it were possible to launch denial-of-service attacks using stack space, the same technique of bytecode engineering could be applied to restrict stack consumption. Although in the current runtime, limiting stack space on a per thread basis is dealt within the JVM itself.

5.3 SYSTEM LIMITATIONS

Although the system works fine in most usual cases but there is a way in which the accounting can be deceived. This can be done by reincarnating the object within the finalizer method after the de-allocate notification has been sent to the memory manager. To accomplish this the object can pass its own reference to some other location which prevents the garbage collector from trashing it even after it has called the finalizer on this object.

To prevent this from happening and for the sake of fair accountability, we have decided to disallow finalizers in un-trusted code. To accomplish this task a class within the un-trusted code containing a finalizer can be prevented from being loaded into the system.

6. EXPERIMENTS AND RESULTS

In this section we evaluate the performance of the system by conducting a few tests to monitor memory consumption by threads running in a system.

6.1 ACCURACY

Several experiments were conducted to test the accuracy of the memory management system. Each of these experiments demonstrates the execution of a dominant thread that allocates objects of known sizes at known rates. Figure 5 shows the actual memory consumption of a thread compared to its account with the memory manager, plotted against time. This instance shows that the accountability system is detecting allocations and de-allocations with extreme precession. This is clearly visible via regions of significant overlaps between the two graphs in figure 5. This precision owes to the fact that the thread being monitored here allocates only non-system objects and primitive arrays.

Figure 6 shows a similar comparison for another thread monitored by the system. This thread allocates both non-system and system objects with second level allocations. In this instance it can be seen that the actual memory usage continues to increase whereas the accountability interface is only able to account for only a fraction of the thread’s consumption. This is a demonstration of the phenomenon discussed in Section 5. The memory manager is unable to detect allocations and de-allocations of second level system objects, which results in a huge difference between the actual memory consumption of the thread and the consumption recorded for accountability. Hence the two graphs for actual and recorded consumption are utterly disconnected. Second level system allocations can potentially create a scenario where a thread that has exceeded its allocated limit could continue executing normally.

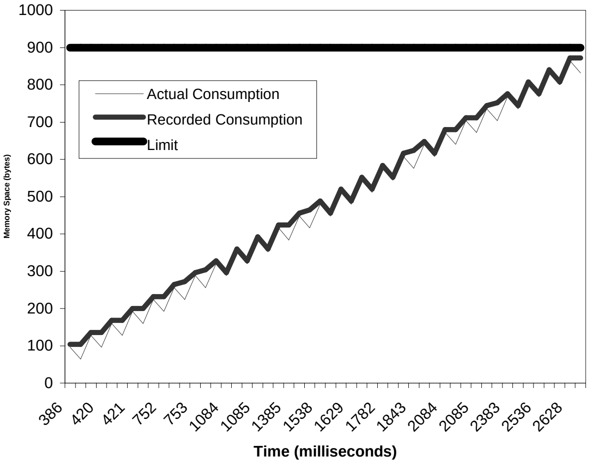

Figure 5. Actual memory consumption vs. Consumption according to accountability data. Non-system allocations.

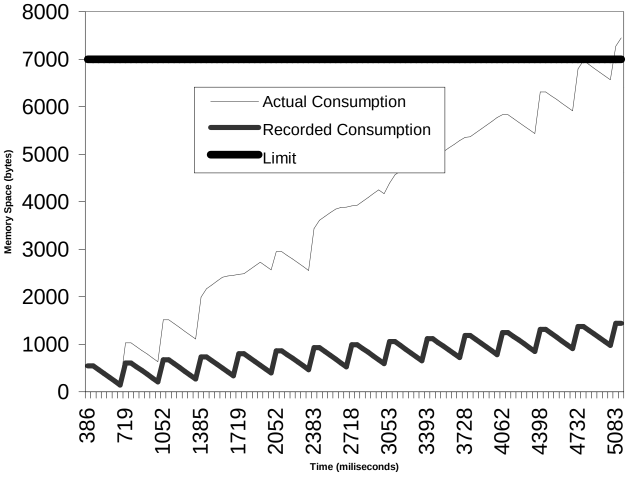

Figure 6. Actual memory consumption vs. Consumption according to accountability data. Multiple System allocations.

6.2 PERFORMANCE

Accounting for heap memory requires additional computation on part of the system. As a direct consequence the system receives a dual performance setback associated with memory accountability. The first delay comes from the extra time it takes to engineer the bytecodes and add hooks for the memory management system. This slowdown is proportional to the size of the file and the number of memory modifying instructions. The second setback comes from the execution of the additional instructions that were added to the classes during bytecode modification. This slowdown is directly proportional to the number of allocating instructions executed during the run of the system. Figure 7 shows a graphical depiction of these performance hits compared to the number of allocating instructions executed in the system. Statistics have shown that execution of a large number of these instructions would generate considerable performance degradation. Fortunately most general-purpose programs have a low percentage of memory allocating instructions.

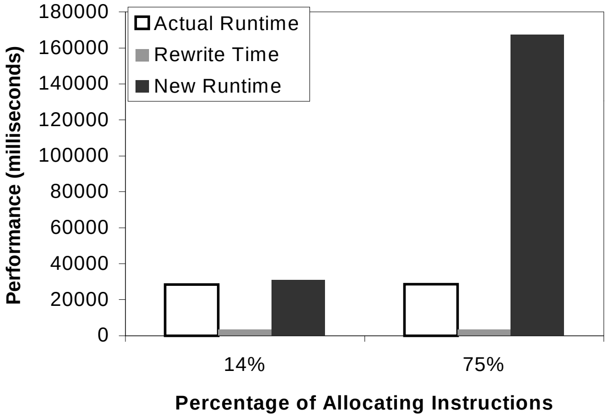

Figure 7. Performance analysis.

7. RELATED WORK

This work is done as part of a complete resource management interface for Java. As discussed earlier, using Java as a runtime environment for deploying web services necessitates resource control and fair distribution of these resources at the system level. In this respect considerable effort has been made to build similar interfaces to manage other resources. Other areas of resource accounting include disk usage, network bandwidth, CPU cycles, write buffers and the cache etc. design of a comprehensive Resource Management Interface has already been proposed for disconnected services [3]. In their proposal the authors have discussed issues regarding disk and cache resources in great detail.

This work is also directly related to systems deploying extensible services. These systems rely on Java for a reliable execution environment. Work has been going on several projects, which have been proposed to run un-trusted code with the system core. Such projects include Active Names [2], Active Networks [6] and Active Services [7], providing services using the potential of active executable content. These systems rely on execution of un-trusted code to provide efficient access to Internet services.

Moreover this work is also associated with efforts made at bytecode engineering. For example successful attempts have been made at runtime optimizations through bytecode modifications by Joseph Hummel [10] and Lars R. Clausen [12]. They present prime examples of bytecode engineering applied to increase the strength of Java, both as a programming language and an execution environment.

An attempt at Resource management for Java was also made through the KaffeOS project [14]. KaffeOS dealt with the resource management issues at the process level but disregarded accountability on a per thread basis. A similar system was Dr. Czajkowski's Jres, which discussed a prototype implementation of a Resource Accounting Interface for Java PL [8]. The Memory Management system described in this paper differs from the one described in Jres with respect to its stricter adherence to the JVM standards and system portability. The prototype that we have described does not dependent on any native code and is built completely at user level.

8. CONCLUSIONS AND FUTURE DIRECTIONS

The strategy of using bytecode engineering to account for heap memory on a per thread basis has been successful partly because of its simplistic low-level design. Unfortunately, this approach becomes a limiting factor when it comes to monitoring the internal system classes, which we have taken as an exception. The goal of building this prototype system is to demonstrate the need for a complete resource management interface for Java. The memory management system proposed in this paper seeks to patch up existing security structure of the language. We have presented a prototype system that uses the technique of bytecode rewriting to build a memory accounting unit on top of the existing JVM model. The intent here is to create a more robust environment for deploying service extensions in which un-trusted code is executed with a minimal overhead. Finally, the issues and performance statistics discussed in this paper would be useful when incorporation of a resource interface is considered for Java.

ACKNOWLEDGEMENTS

We would like to especially thank Dr. Gouda, without whose guidance and moral support this thesis would not exist.

Our appreciation goes to all other people in the area of research. To Dr. Wallach for his experimental insights and for sharing the techniques of bytecode engineering. To Amol Nayate and Bharat Chandra for helping further demystify the correlations within the Active Names System, and distributed web services in general.

We owe a great deal to Usman Shuja and Nabeel Ahmed for their assistance during the project, which helped make this thesis a reality and kept me from giving up.

Last but not least to Dr. Dahlin: Thank you for your patience and continuous support, which helped me in more ways than I can imagine.

REFERENCES

[1] Algis Rudys, John Clements, and Dan S. Wallach. Termination in Language-based Systems, *Network and Distributed Systems Security Symposium* (San Diego, California), February 2001.

[2] A. Vahdat, M. Dahlin, T. Anderson and A. Aggarwal. Active Naming: Flexible Location and Transport of Wide-Area Resources. In Proceedings of the Second USENIX Symposium on Internet Technologies and Systems, October 1999.

[3] Bharat Chandra, Mike Dahlin, Lei Gao, Amjad-Ali Khoja, Amol Nayate, Asim Razzaq, Anil Sewani. Resource Management for scalable disconnected access to web services. WWW10, May 2001.

[4] Chris Laffra. Jikes Bytecode Toolkit. http://www.alphaworks.ibm.com/tech/jikesbt

[5] Dan S. Wallach and Edward W. Felten, Understanding Java Stack Inspection, *1998 IEEE Symposium on Security and Privacy* (Oakland, California), May 1998, pp. 52-63.

[6] David Wetherall, Ulana Legedza, and John Guttag. Introducing New Network Services: Why and How. In IEEE Network Magazine, Special issue on Active Programmable Networks, July 1998.

[7] Elan Amir, Steven McCanne, and Randy Katz. An Active Service Framework and its Application to Realtime Multimedia Transcoding. In Proceedings of SIGCOMM, September 1998.

[8] G. Czajkowski and T. von Eicken. Jres: A Resource Accounting Interface for Java. In Proceedings of 198 ACM OOPSLA Conference, October 1998.

[9] Gosling, J., Joy, B., and Steele, G. The Java Language Specification. Addison-Wesley, Reading, Massachusetts. 1996.

[10] Joseph Hummel, Ana Azevedo, David Kolson, and Alexandru Nicolau. Annotating the java bytecodes in support of optimization. Technical Report ICS-TR-97-01, University of California, Irvine, Department of Information and Computer Science, April 1997.

[11] Jon Meyer, Troy Downing. Java Virtual Machine. O'Rielly. 1997.

[12] Lars R. Clausen. *A java bytecode optimizer using side-effect analysis*. Concurrency: Practice and Experience, November 1997.

[13] Mike Dahlin, Bharat Chandra, Lei Gao, Amjad-Ali Khoja, Amol Nayate, Asim Razzaq, Anil Sewani. Using Mobile Extensions to Support Disconnected Services. Technical Report TR-2000-20, University of Texas at Austin.

[14] Godmar Back, Wilson C. Hsieh, and Jay Lepreau. *Processes in KaffeOS: Isolation*, resource management, and sharing in Java. In Proceedings of the 4th Symposium on Operating Systems Design and Implementation (OSDI 2000), San Diego, California, October 2000. USENIX Association.