Resource Management for Extensible Internet Servers

Grzegorz Czajkowski, Chi-Chao Chang, Chris Hawblitzel, Deyu Hu, Thorsten von Eicken
Department of Computer Science, Cornell University
Ithaca, NY 14850, USA
{grzes,chichao,hawblitz,hu,tve}@cs.cornell.edu

1 Introduction

With the spread of the Internet the typical computing model for servers is undergoing a drastic change. In the past, server systems have moved from providing interactive time-sharing service to providing fileserver and now more general back-office (mail, database, web, etc.) services. While the characteristics of the new Internet server systems are not yet clear, we expect that Internet servers will have at least three characteristics that distinguish them from today's servers: (i) high code mobility, (ii) large numbers of anonymous users, and (iii) significant concern for the efficient use of resources.

Due to the large number of users of popular Internet sites, the sophistication of the services that can be offered is primarily limited by the available computing resources and by security concerns. In this paper, we focus on the resource management issues arising in extensible server systems that must support mobile, untrusted code and large numbers of users.

The primary goal of the resource management we propose is to take advantage of resource consumption tradeoffs. In traditional server systems, the administrator decides which services run on which machines. Databases, for example, use three-tier architectures such that request validation, database processing, and data formatting can be configured to occur on different server machines. The administrator also decides how to split computation between the servers and the client machines. The lure of the Internet combined with portable safe language technology (such as Java) is that these tradeoffs can be made much more dynamically and can be customized more easily. Basically, this is possible because it is increasingly feasible to move code around on demand.

In order to take advantage of these tradeoffs, however, the programs need information on their current resource consumption, indications on the available resources in the systems, as well as incentives to use the resources leading to the highest overall throughput. We claim that in many situations the response of a service to resource shortage does not have to be a degradation of quality. Instead, it is often possible to trade one resource for another one and continue to provide similar quality of service.

In this paper we focus on extensible, Java-based servers, which support the execution of untrusted code, called *servlets*[1]. The servlets can be stored on the server itself or they can be uploaded dynamically by users in order to customize the server functionality. Even though an initial work presented here has been done in the context of standard Java, our target environment is the J-Kernel [4]. We discuss how we extend the J-Kernel to provide resource usage feedback as well as incentives for servlets to adapt their behavior in response to load fluctuations.

An important feature of the J-Kernel is that it executes within a single Java Virtual Machine (JVM). As is the case of many other safe language environments, Java does not provide mechanisms to constrain resource consumption of applications, beyond the simple manipulation of thread priorities. However, using Java (or any other safe language) to implement extensible server environments requires the ability to limit resource consumption. We have developed JRes (pronounced "*jay-res*") - a resource accounting and limiting interface for Java - in order to account for and impose limits on resources available to particular servlets.

We also claim that short-term guarantees are essential in enabling resource usage tradeoffs: if a servlet decides to change its behavior in order to take advantage of an under-utilized resource, the system should

[1] The first servlet architecture and implementation was proposed in the Java Web Server [6]. The J-Kernel differs significantly from the Java Web Server in its emphasis on protection and efficiency.

be able to guarantee the actual availability of the resource for some short period of time. This is in contrast to long-term guarantees which seem to lead to high costs because they inevitably require overly conservative reservations. Finally, we argue that economics-based models are not really applicable in dynamically extensible Web servers: while these models are currently fashionable, the actual implementations remain unconvincing and do not appear to map onto real user behaviors.

2 The target environment

The environment we propose is based on the architecture of the J-Kernel [4]. The J-Kernel is a μ-kernel that executes on a single Java Virtual Machine (JVM) and provides multiple protection domains in that JVM. The J-Kernel is written entirely in Java and provides a new class loader which allows servlets to be uploaded and executed each in its own protection domain. Protection in the J-Kernel is based on software capabilities. Each protection domain (i.e. a servlet) has a namespace that it controls as well as a set of threads. Cross-domain calls are performed by invoking methods of capabilities obtained from other domains. The J-Kernel's class loader interposes a special calling convention for these calls as follows: arguments and return values are passed by reference if they are also capabilities, but they are passed by copy if they are primitive types or non-capability objects (deep copy in this case). The effect is that only capabilities can be shared between protection domains and references to regular objects are confined to single domains.

The J-Kernel was carefully designed to facilitate resource management. An important decision made is to disallow the direct sharing of objects between domains, i.e. two domains cannot both have a reference to the same object. The rationale is that sharing objects arbitrarily makes resource accounting problematic (who should be accountable for an object shared by many servlets?) and resource reclamation and domain termination become difficult at best (see [4] for details).

In the proposed environment, the task behavior and resource consumption assumptions may be different from those valid for traditional workloads because of the large volume of tasks and high network utilization. First, even though the execution of servlets will be triggered by independent users, it is likely that very large numbers of servlets will be identical, be it a data mining agent, a set of image manipulating methods, or a custom program that handles electronic submission of documents. This is caused mostly by the fact that very few users in the real world actually write programs themselves — from operating systems to text editors to databases, users rely on code delivered by a relatively small number of software vendors for any given applications niche. Executing large number of similar tasks simultaneously can make the system unstable with respect to resource needs - several hundred copies of a CPU-bound application can stress the processing power of any system. This creates both the need and the opportunity for utilizing resource tradeoffs.

Another difference is the relative need for different resources, when servlets are compared to most applications executed in traditional environments. We expect that servlets will require more network bandwidth per CPU cycle due to the inherent communication between a servlet and its remote clients. Memory needs will most likely be no different than for "traditional" applications, while CPU time consumption is likely to be low for a vast majority of servlets but very high for a few.

Even though this work is focused on the single JVM case, it is important to note an inherent distributed tradeoff to be exploited in the proposed setting: one can almost always trade CPU time on the server for the network bandwidth and CPU time on the client.

3 Resource management mechanisms

The resource management for J-Kernel is based on the JRes interface [2]. The interface allows accounting for resource consumption on per-thread and per-thread group basis[2]. The resources accounted for are CPU time, heap memory and network traffic. The implementation of JRes contains a small native component but is mostly based on Java bytecode rewriting. JRes works both in a traditional, security manager-based Java system, and in a capability-based system like the J-Kernel.

---

[2] The notions of threads and thread groups map well onto the J-Kernel-specific notions of thread segments and protection domains. For simplicity, the subsequent discussion will focus on managing resources on per-thread basis.

The overheads of accounting for memory consumption are within 20% of the execution time of an unmodified Java program[3]; accounting for network traffic adds less than 5% to roundtrip latency time over a local Ethernet; and accounting for CPU results in less than 4% of execution time overhead. These overheads could be drastically reduced if the functionality of JRes were built into the JVM.

In addition to accounting for resource usage, JRes allows Java-based execution environments (browsers, extensible servers) to be notified of thread and thread group creation. Each such notification can then be used to impose resource usage limits and to set *overuse callbacks*, which are actions to be taken whenever a limit is exceeded. Finally, JRes allows applications to query the system about the load on particular resources.

The current limitations of JRes are: resource ownership cannot be transferred, memory and network resources consumed by native code cannot be tracked, and the fact that JRes is not scheduling threads by itself. We are currently extending JRes to allow resource ownership transfer. Dealing with native code can be partially addressed by incorporating JRes into the JVM. managing thread scheduling requires either redesigning the JVM so that it does not rely on kernel threads (the most current JVM from both Sun and Microsoft map Java threads one-to-one onto kernel threads) or changing the thread scheduler of the underlying operating system.

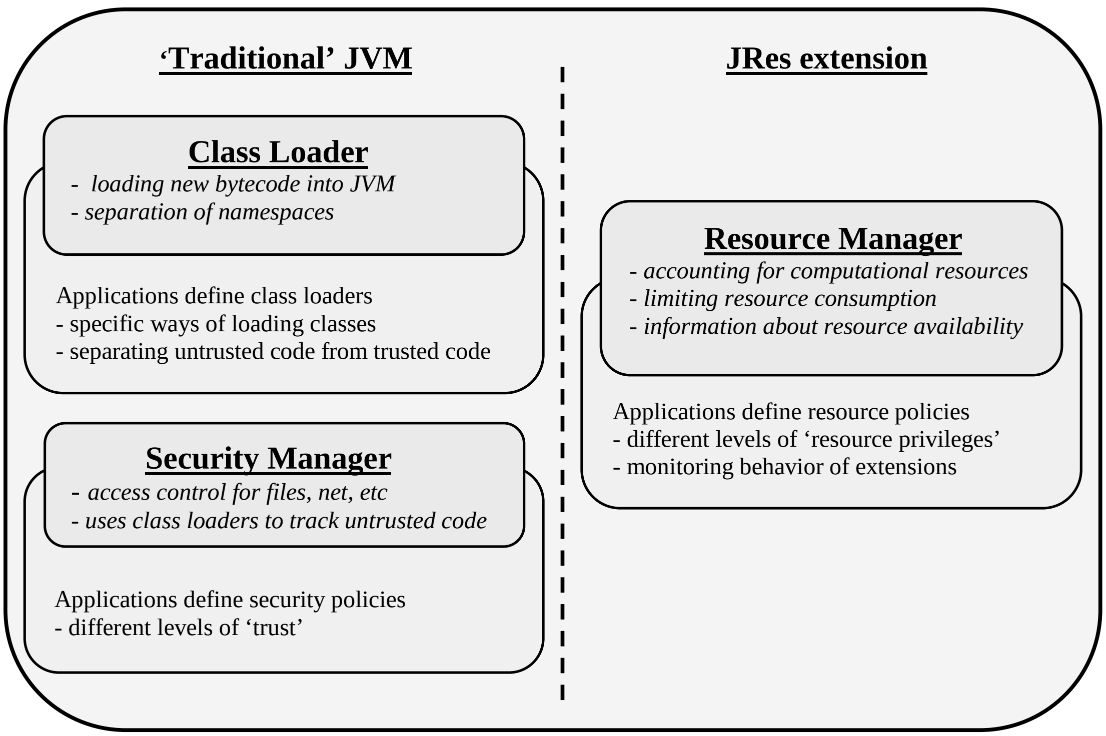

Figure 1. Extending the JVM with resource managing facilities.

JRes can be viewed as extending the JVM with an important component: a resource manager. Figure 1 shows what features the resource manager adds to the JVM [1]. We claim that for extensible systems the presence of a resource manager is as vital as the class loader and the security manager. A similar conclusion with respect to the importance of resource management in Java has been reached by another group [9].

[3] The test suite consisted of JavaCC, Jigsaw, Caffeine Marks and a Lisp interpreter. The programs were executed on a 300 MHz Pentium II PC with 128 MB of RAM under Microsoft's Visual J++ version 1.1.

Figure 2 shows how JRes can be used to limit resources available to threads (a very similar example can be written to limit resources available to thread groups, domains, etc). The `SimpleResPolicy` can be thought of as being used inside a Web browser. Its goal is to limit available CPU time to 500ms per thread. Whenever this limit is exceeded, a pop-up window is created to ask the user if the offending applet thread should be killed or given more CPU time. A slightly longer piece of code can implement a policy in which every applet thread group gets no more than 100ms of CPU time out of every second of wall-clock time.

In addition to accounting for and limiting resource usage, JRes could be used to provide some resource guarantees. This is possible because JRes keeps track of all active threads, enforces resource limits and cooperates with the class loader, controlling the acceptance of new servlets to be started. Providing an adequate interface and infrastructure for requesting and obtaining guarantees in the context of Java and JRes is being currently investigated.

```
class SimpleResPolicy
      implements ThreadRegistrationCallback, OveruseCallback {
  private Object cookie;

  private SimpleResPolicy(Object cookie) { this.cookie = cookie; }

  public static synchronized void initialize() {
    Object cookie = new Object();
    ResourceManager.initialize(cookie);
    SimpleResPolicy srp = new SimpleResPolicy(cookie);
    ResourceManager.setThreadRegistrationCallback(cookie, srp);
  }

public void threadRegistrationNotification(Thread t) {
    if (t.getThreadGroup().getName().equals("system")) { return; }
    ResourceManager.setLimits(cookie, RESOURCE_CPU, t, 500, this);
  }

 public void resourceUseExceeded(int resType, Thread t, long value) {
    if  (resType == RESOURCE_CPU) {
      ThreadFate.popupMessage(msg, t);
    }
  }
}
```

Cookie is used for authentication.

Called whenever a thread is created.

Limit: 500ms of CPU time.

Called whenever an applet uses more than 500 ms of CPU time.

Users are given a choice: kill offending threads or give them unlimited resources.

Figure 2. Simple resource management policy: no thread can use more than 500ms of CPU time. The class `ResourceManager` and the interfaces `ThreadRegistrationCallback` and `OveruseCallback` are defined in the package `cornell.slk.jres`.

4 Short-time resource guarantees

In the proposed Internet computing model we claim that providing short-term resource availability guarantees is preferable to providing no guarantees at all or to providing more traditional long-term guarantees. Without guarantees it is difficult to utilize resource tradeoffs, because no central coordinator ensures that decisions taken will be profitable and result in a stable system. Long-term resource reservations and guarantees tend to unnecessarily overbook resources and lead to the under-utilization of the system. In this section we discuss the need of providing short-term guarantees and the consequences of such a decision.

Our approach addresses the issue of how to utilize the unused resources with the most benefit for the large population of servlets. In particular, we advocate providing servlets with resource availability feedback, based on resource usage accounting and providing means by which servlets can require and obtain various short-term resource- and performance-related guarantees. The first postulate means that every servlet should be able to query the environment for the following information: (i) how much resources it is using, (ii) how much guaranteed resources, beyond its share, it could obtain, and (iii) for how long. The second item allows servlets to plan ahead and decide, which of possible resource tradeoffs (if any) should be taken advantage of. Combined, resource information feedback, guarantees, and an opportunity for faster execution provide an enabling mechanism as well as an incentive for utilizing these tradeoffs.

As argued earlier, an Internet server should support large numbers of simultaneously executing servlets. By observing the most common way typical Internet users interact with the Web servers, we conjecture that a typical servlet will have a rather long lifetime (similar to the duration of a Web-browsing session) and will process relatively short-lived requests (similar to Web page fetching requests). Short-term resource guarantees suffice for the whole lifetime of such a request. Let us consider a typical execution scenario, in which a servlet has to process a simple request. The servlet queries the server about available resources. Based on this information the optimal choice of the request processing algorithm is made by the servlet, which then obtains resource guarantee and finally processes the request.

In the case of Internet servers which satisfy our workload assumptions, the rule of optimizing systems for the most common case suggests providing short-term resource guarantees, since most requests are short-lived. An application that is neither a loop processing mostly short-lived requests nor a collection of loosely connected tasks has to change its behavior in order to exploit short-term resource guarantees. If it is impossible to restructure a program to belong to one of the two classes of applications named above, the program may try to adapt to changing resource supply via exploiting resource tradeoffs on a short-term time scale. Conforming to this model may be easy for some applications; for instance, a parallel ray tracer might adjust the level of parallelism continuously during rendering of an image. This decision could be based on processor utilization, number of other threads in the system, etc. On the other hand, converting some applications, like *uuencode*, to utilize short-term resource guarantees may be difficult.

Utilizing resource tradeoffs goes hand in hand with guarantees that the server should give to servlets. Changing behavior from CPU-bound to network-bound, without a guarantee that at least for a short time this will actually pay off is risky at best. Due to the very dynamic nature of the whole environment, the only long-term guarantee that a servlet can obtain is a guarantee of fair share. However, since the number of servlets changes, even this weak guarantee cannot be translated into absolute quantities. In the presence of large numbers of servlets and variations in resource load, the only guarantee a servlet may count on is the short-term behavior of the system.

Long term guarantees are both impractical in such a system due to expected high servlet traffic and expensive, since they typically tie up more resources than actually needed for long periods of time. Resource-aware applications can avoid over-reserving resources for long periods of time. A way to do this is to request small amount of resources, just necessary for providing basic quality of service (the smaller such a request, the more likely it is to be granted). At runtime, exploiting short-term variations in supply and demand for various resources can lead to much better performance than the baseline. The system should provide the necessary information so that servlets can adjust their behavior to take advantage of possible resource tradeoffs, as described below.

We do not claim that long term resource availability guarantees are not practical in general. Recent work [7,8] demonstrates the need for providing such guarantees and the solutions for delivering them. Our goal is to understand what can be done in a system in which only short term guarantees are possible.

5 Advantages of being resource aware

A simple experiment has been conducted in order to gain initial confidence in exploiting resource tradeoffs. A Java application (referred to from now on as *server*) sends a large text document to eight clients, each of which is located on a separate computer. All participants are within the same local, Fast Ethernet-connected network. The clients are single processor P6 200 MHz machines; the server resides on a dual processor P6 200 MHz computer. The server starts eight threads, each handling a different client. The threads are started within two-second intervals. Every new thread can choose to send the text "as is" or compress it before sending, using `java.util.zip`.

In the first run of the system, all the connections are forced to opt for uncompressed sending. This resultes in effective out-bandwidth (total amount of document data leaving the server) of about 8.1 MB/s and processor utilization of about 45%. In the second run all connections are compressing data before sending them, which results in effective bandwidth of about 7.3 MB/s and processor utilization of 100%. In the third run, before starting the threads the connections can query the system, using the JRes interface, about the CPU load. If the CPU load is below 99%, a new thread will compress data before sending, since the CPU is not totally loaded yet; otherwise, the data will be sent without having been compressed. This leads to three connections using compression, and five connections sending uncompressed data. The achieved effective bandwidth is 11.7 MB/s and the CPU load is 100%.

Of course, this experiment is very simple but is a convincing indication of the possible benefits that may be uncovered while exploiting resource tradeoffs. A difficult issue that needs to be addressed when dealing with resource tradeoffs is comparing availability of different resources. For instance: how to compare available CPU with available network in order to decide which tradeoff to pursue? This question is nicely answered in a system where every resource has a "price". Despite this advantage of using economics-based models for resource management we have decided to abandon this approach, as explained in the next section.

A final point about resource-awareness of applications is that providing resource-sensitive standard libraries plays an important role in enabling resource-aware applications. In such libraries different implementations of the same functionality exist, each of them with different resource consumption and execution times. For instance, popular core Java classes implementing hashtables and vectors of objects can be implemented either with fast access time but low memory requirements or with reverse properties. Similarly, two different versions of sockets may exist: a standard one and one which transmits and receives compressed data.

6 A critique of economics-based approach to resource management

A number of research projects advocate the use of economical models to force applications to compete for resources [3,5,10]. This approach may seem ideal for Internet systems. In fact, there is an unquestionable merit to bidding for resources – prioritizing applications according to the amount of funding they have, which should in turn reflect the relative importance of different tasks. Another advantage is the ability to compare the 'value' or 'importance' of two resources through comparing their 'price'. However, there are several problems with this approach. First, the actual pricing mechanism implemented in existing and proposed systems seems to invariably be either cheap to implement but rather arbitrary [5] or more realistic auction-based but with high-overheads. For example, Edell et al. [3] propose a system that enables billing users for their TCP traffic but it increases the average connection setup time by 66%. A second issue is who actually "funds" applications in economy-based resource management environments. If it is the system, then the accumulation of "money" over periods of inactivity must be dealt with. Some systems propose a form of taxation [5], but the formulas used are again rather arbitrary. If the user funds the applications based on real money the pricing and its evolution must be easy to understand. Most current consumer services appear to quickly converge on flat rates. However, introducing flat rates actually defeats the purpose of economics-based models. If the user funds the applications through "funny money" it may just not be taken seriously, since the user will simply not feel the penalty for misspending or misbehaving.

The problems outlined above made us resign from adopting a system-wide economic approach. Resources used by applications are accounted for, but we leave billing up to the service providers themselves. Applications are given information about their own resource consumption and can change their behavior accordingly. The incentive for doing so is an opportunity to improve performance.

7 Summary

The purpose of this work is to understand resource management for a quickly emerging model of Internet servers computing. The main characteristics of this model are high code mobility and large numbers of anonymous users, which imply different resource demands of applications when compared to traditional operating systems. Efficient use of distributed resources becomes a must if the model is to be useful. We advocate focusing on resource-aware applications and providing short-term resource guarantees. Exploiting changes in resource availability may prevent degradation of quality of service to end users in case of shortages of a particular resource. We criticize economics-based approaches to resource management as inadequate for the envisioned system.

A prototype environment, based on the J-Kernel and JRes, has just become operational. The proposed system support consists of resource accounting and limiting facilities, means of querying the system about resource availability, and providing short-term guarantees of resource availability. Future plans include extending the functionality of the prototype and further experimentation.

8 Acknowledgements

This research is funded by DARPA ITO contract ONR-N00014-92-J-1866, NSF contract CDA-9024600, a Sloan Foundation fellowship, and Intel Corp. hardware donations.

9 References

1. Arnold, K and Gosling, J. *The Java Programming Language.* Addison-Wesley.

2. Czajkowski, G, von Eicken, T. *JRes: A Resource Accounting Interface for Java.* Proceedings of the ACM Conference on Object Oriented Programming Systems, Languages and Applications. Vancouver, Canada, October 1998.

3. Edell, R, McKeown, N and Varaiya, P. *Billing Users and Pricing for TCP*. IEEE Journal on Selected Areas in Communications, Vol. 13, No. 7, September 1995.

4. Hawblitzel, C, Chang, C, Czajkowski, G, Hu, D. and von Eicken, T. *Implementing Multiple Protection Domains in Java.* Proc. Annual USENIX Conference, New Orleans, LA, June 1998.

5. Heiser, G, Lam, F, and Russel S. *Resource Management in the Mungi Single-Address-Space Operating System*. Proc. 21st Australasian Computer Science Conference, Perth, Australia, February 1998.

6. Java Web Server. http://java.sun.com/products/java-server/webserver/index.html.

7. Jones, M, Rosu, D and Rosu, M-C. *CPU Reservations and Time Constraints: Efficient, Predictable Scheduling of Independent Activities.* Proc. 16th Symposium on Operating Systems Principles. Saint-Malo, France, October 1997.

8. Nieh, J and Lam, M. *The Design, Implementation and Evaluation of SMART: A Scheduler for Multimedia Applications.* Proc. 16th Symposium on Operating Systems Principles. Saint-Malo, France, October 1997.

9. Thullman, P and Lepreau, J. *Nested Java Processes: OS Structure for Mobile Code.* Proc. 8th ACM SIGOPS European Workshop, Sintra, Portugal, September 1998.

10. Waldspruger, C. A *Spawn: A Distributed Computational Economy.* IEEE Transactions on Software Engineering, 18(2):103-117, February 1992.