Enhancing Java Server Availability And Performance With JAS

Reinhard Klemm, Navjot Singh
Bell Laboratories, Lucent Technologies
600 Mountain Ave.
Murray Hill, NJ 07974, U.S.A.
Email: {klemm,singh}@research.bell-labs.com

Abstract

The Java programming language is increasingly being used in the implementation of servers that have stringent availability, reliability, and performance requirements. The Java Application Supervisor (JAS) software can be attached to any Java program to supervise its execution, i.e., JAS automatically detects and resolves certain reliability and performance problems according to user specifications. JAS does not require source code modifications or recompilation of the supervised Java program. Thus, JAS can be used to enhance the availability of Java servers and decrease the risk of performance degradation with very little effort on the developer’s part. In an experiment with a Web proxy that exhibits reliability and performance problems under heavy load, we demonstrate an increase in the rate of successful requests to the server from 61% to 94% by using JAS and a decrease in the average request processing time by 22%. JAS has also been used successfully in two Java servers at Bell Labs to monitor server reliability and performance and ensure long-term availability. JAS is lightweight in the sense that it typically imposes very little execution time and memory overheads on the supervised program.

Keywords: Availability, Java, Performance, Program Supervision, Reliability, Servers.

1 Introduction

The Java programming language is increasingly being used in the implementation of programs that used to be the domain of more traditional programming languages such as C and C++. This also applies to servers with stringent reliability and performance requirements, e.g., in the telecommunications and e-commerce world. Quite often and for a variety of reasons, performance and reliability problems materialize only after a program has been deployed to the user. Thus, a program with stringent performance and reliability requirements ideally comes with built-in software mechanisms that detect performance and reliability problems at runtime. These mechanisms can alert a supervisor or automatically attempt to resolve detected problems. At the very least, they can record reliability and performance trouble spots in the program and thus help developers to improve the program. Let us call a collection of such mechanisms an application supervisor and the act of detecting and resolving performance and reliability problems during the execution of a program application supervision. Because Java already has a sophisticated policy-based security model for application execution, we do not extend our notion of application supervision to execution security. If an application supervisor is an integral part of a program (target), we call it an internal application supervisor. If the supervisor is attached to the target in such a way that the latter can execute without the supervisor, i.e., if the supervisor is not an integral part of the supervised program, we call it an external application supervisor.

Application supervision is intended to detect performance and reliability problems that were overlooked or not anticipated during software testing and maintenance phases and that show up
during the operational phase of the program. Application supervision itself should be lightweight in the sense that the execution time and memory overheads imposed on the target program are small. This is an ambitious goal for the designer of an application supervisor. At the same time, an application supervisor does not contribute to the functional purpose of the program. Therefore, many software projects avoid the time, expense, and required expertise associated with building a supervisor. To free developers from the burden of building an application supervisor, we designed and built JAS (Java Application Supervisor). It is an external application supervisor that is reusable across nearly all Java target programs simply by changing its configuration. Moreover, the same target program can be deployed to different users with different JAS configurations, thus conveniently allowing different types of fault detection and recovery for different usage scenarios. JAS does not require changes to the target source code or bytecode or a recompilation of the target, i.e., JAS is non-intrusive. While JAS cannot eliminate the need for target-specific application supervision, it can very often reduce the time- and effort-intensive implementation of mechanisms that monitor and handle reliability and performance problems. It can also serve as an additional safety net for programs that come with application-specific fault-tolerance mechanisms. The execution time and memory overheads added by JAS to the supervised application are typically under 10%.

JAS can very quickly detect problems and, if necessary, restart the target application or a specific thread. If the target application is a server, this amounts to a brief interruption of the services it offers to clients. In this way, JAS can significantly improve the availability and long-term performance of the target application. We will quantify the availability and performance benefits resulting from using JAS by describing an experiment in this paper. In this experiment, we used JAS to supervise the execution of the Jigsaw Web proxy from the World Wide Web consortium. Jigsaw sometimes exhibits a faulty behavior when processing a large number of browser requests simultaneously. By instructing JAS to restart Jigsaw whenever JAS detects a faulty behavior of Jigsaw, the number of successfully processed requests increased from 61% to 94% in our experiment. At the same time, the average time it took to complete a client request decreased by about 22% with JAS. In this article, we also describe the use of JAS in monitoring two telecommunications servers at Bell Labs.

For JAS to be useful, its configuration for a specific target must be much simpler than to program a customized application supervisor. The user configures JAS by either adopting or changing a set of default policy templates in a configuration file that JAS automatically generates from the bytecode of the target application. Each policy template is associated with a specific aspect of reliability or performance of the entire target application or a subclass of its threads or objects. Once a policy template has been parameterized, it becomes a policy that specifies how JAS should react to a problem once it has been detected. To understand the following examples, notice that a graphical frontend can be attached to JAS that can visualize some aspects of the state of the supervised application and the detected problems. An example of a reliability-related policy would be (in prose; the actual JAS policy is more formal and concise):

Notify frontend whenever a thread of class ClientRequestHandler dies abnormally. Restart application if more than 5 threads of class ClientRequestHandler die abnormally within 90000 ms.

An example of a performance-related policy would be (in prose):

If the garbage collector runs more than three times within 300000 ms for more than 10000 ms each time then restart the application once it has become idle.

JAS can be of service even if the user does not have knowledge of the internal structure of the target application. For example, the user can almost always assume that an uncaught exception in
a thread, leading to the immediate death of the thread, constitutes a software failure and warrants some action such as restarting the application. However, the more the user is familiar with the internals of the target application the more the user can tailor the JAS configuration and thus the more precise the application supervision can be.

This article is organized as follows. Section 2 discusses some related approaches to external application supervision. The features of JAS are described in detail in Section 3. Section 4 explains how to set up JAS for supervising a given target application. Some aspects of the JAS architecture and implementation are presented in Section 5. We describe some experiments showing the effectiveness of JAS for real applications in Section 6. This section also measures the resource overhead imposed by JAS on a target application. The paper concludes with a brief description of work in progress and future work in Section 7.

2 Related Work

JAS combines aspects of software fault tolerance, performance monitoring, on-line as well as off-line debugging, and software testing. Software fault tolerance is effected by software that detects software failures and recovers from them. For example, a process that periodically checks via the operating system whether another process is still alive and, if not, restarts the second process, provides software fault tolerance. Because JAS targets only Java applications and because of its design, JAS achieves a unique set of goals that distinguishes it from other solutions:

- external application supervision for nearly all Java applications
- no access to or recompilation of the target application required
- low performance overhead and low memory overhead
- simple configuration
- detection of a large number of reliability and performance problems
- automatic resolution of some detected problems

Some details about JAS not covered in this paper can be found in [8]. The following is a description of some other approaches that are more or less closely related to JAS.

A good overview of the architecture and analysis of software fault tolerance mechanisms can be found in [10]. In addition to performance problem detection and resolution, JAS provides software fault tolerance in the application layer as defined in [5]. While [5] detects and recovers from process hangs and crashes, JAS has problem detection and resolution capabilities that go far beyond dealing with process crashes and hangs. JAS provides software fault tolerance partially by automatically providing default exception handlers for the target application. A similar concept was presented in [1] with the difference that the default exception handlers described in [1] are installed by the compiler implying that the target source code has to be available for this solution.

There is a myriad of commercial Java debuggers and profilers as well as Java black-box and white-box testing tools. These programs are used during the software development and maintenance phases. JAS can be used to test certain aspects of programs but is really designed to accompany the target application once it has been deployed to the user. It does not require test case generation and works behind the scenes during the normal execution of the target. Thus, JAS captures reliability and performance problems that arise with the user's actual input data and computing environment. Moreover, JAS can automatically resolve many detected problems and thus extend the uptime of a long-running program.

The need for software-implemented application supervision was recognized already in the early 1980s. In [13], the reader will find the description of a supervisor for real-time applications called a real-time monitor. Similar to JAS, the real-time monitor bases the detection and resolution of
performance and reliability problems on user-specified policies that are, however, much more flexible than those allowed in JAS. The difference is that the functionality of and the real-time constraints for the real-time monitor require special hardware support and the need for two machines as well as access to the source code of the target application.

An application supervisor for the ADA programming language is presented in [3]. Like JAS and the real-time monitor in [13], the ADA supervisor lets the user specify policies consisting of predicates and actions. Policies for the ADA supervisor are more flexible than those for JAS. On the other hand, the ADA supervisor requires source code editing of the target program and subsequent recompilation. The authors of the ADA supervisor also report an unquantified high performance overhead.

The Meta system [11] is a toolkit for writing external application supervisors for distributed target applications written in C and running on Isis and Unix. Meta allows the placement of generic and customized sensors (probes into the target that can be queried and deliver information about the program or the program state) and actuators (code that affects program execution) and is thus very powerful. Unlike JAS at the current time, Meta allows the coordinated supervision of all remote components of a distributed application. However, using Meta requires detailed knowledge of, changes and additions to, and recompilation of the target source code. Furthermore, the structure of the target has to be modeled by the Meta user in a special-purpose language (Lomita) in order to have Meta supervise all components of a distributed application.

SwiFT and NT-SwiFT [4], designed in our department at Bell Laboratories for C and C++ programs, are software collections that automate error detection and recovery, checkpointing, event logging and replay, communication error recovery, incremental data replication, IP packet re-routing in case of server failures, and other software fault tolerance mechanisms. At this point in time, JAS focuses on error detection and recovery and event logging only. SwiFT and NT-SwiFT can detect process crashes and, if the target application is modified accordingly, also process hangs. JAS, on the other hand, targets only Java programs and, in return, can detect a far wider range of reliability and performance problems, down to the application thread- and object-level if desired by the user. JAS never requires target program modification or recompilation. Moreover, JAS allows the user to override some of the target application behavior in order to avoid reliability and performance problems. For example, JAS can be configured to automatically limit thread creation in the target to a certain number of threads. Whenever the target attempts to exceed this number, JAS will suspend newly created threads until the total number of threads has fallen below the specified threshold.

There are several research projects that allow automatic modifications to Java bytecode at or before class load time according to user-supplied specifications. Examples of such tools are BIT [9] and JOIE [2]. The user of a bytecode modification tool writes bytecode transformation rules in a language such as Java, and the tool generates a modified class file that contains the user-specified bytecode additions and changes to the original class file. While this approach is very powerful, it requires from the user a profound understanding of Java class files and of the target application, whereas JAS requires neither. In addition, JAS can perform tasks that are not possible with bytecode transformations such as monitoring the garbage collector in the Java virtual machine.

Some commercial products such as FullTime Cluster [14] provide features similar to JAS but the resolution of their policy parameters is not the application-internal level and rather the application or computer cluster node level. Accordingly, policy parameters in these systems are not objects, threads, classes, and other application-internal entities but entire applications, nodes, and application or node parameters such as CPU and memory load. Actions, therefore, do not apply to application-internals either but to entire applications or cluster nodes.
3 JAS Features

JAS is a lightweight external supervisor for Java applications. Its problem detection and resolution capabilities cover performance and reliability aspects of the application and are completely transparent to the application, i.e., the functionality of the application is not affected by JAS and no source code or bytecode change or recompilation is necessary. The current JAS implementation can supervise any Java application that fulfills all of the following conditions:

- its functionality does not depend on the total number of threads in its address space (because JAS adds a thread to the supervised application)
- it makes no assumptions about the order in which threads are scheduled or about absolute times for the execution of code
- it does not change the implementation of certain Java API classes listed in Section 5
- it is executed on a JavaSoft Java Virtual Machine 1.2 or higher

The following reliability-related events can be detected by the current JAS implementation:

- virtual machine shutdown
- illegal number of threads (e.g., spawning the same thread more than once can indicate a programming bug)
- application hung, i.e., execution time exceeds user specification
- thread hung, i.e., execution time exceeds user specification
- thread terminates (but is not supposed to ever terminate)
- application exits with System.exit(n) where n is different from a user-specified number that would indicate a normal application termination (typically, n = 0)
- thread terminates abnormally, i.e., due to an uncaught exception
- exception thrown

The Java equivalent of a C/C++ application crash is usually a thread or application termination due to an uncaught exception, ranging from NullPointerExceptions to VirtualMachineErrors. Some Java programs catch a variety of exceptions but do not deal with them other than printing or logging the exception, thus leaving the application in an illegal state. JAS can detect exceptions whether they are handled by the application or not. In the former case, JAS allows supplementing the exception handler in the application by responding to the corresponding exception with one of the actions listed below before the application exception handler is executed.

The current implementation of JAS can detect the following performance-related events in the target application:

- garbage collector runs too often
- garbage collector runs for too long
- number of threads exceeds threshold

Currently, JAS can respond to a detected problem in the following ways:

- ignore event
- notify frontend and log problem report
- suspend additionally spawned threads (if number of threads in target exceeds a threshold)
- restart thread
- restart idle application
- restart application immediately
- quit application
The frontend notification can be combined with every other action. The restart thread action restarts the run method of a thread class with all the object-level variables of the thread object intact. Depending on the application, this could corrupt the internal state of the target or could otherwise result in unexpected system behavior. Thus, this action has to be used with caution. The thread restart action typically works well for threads that have little or no interaction with other threads. Client request handlers are a good example of such threads. The restart idle application action restarts the application once all threads in the target have not consumed CPU time for a certain user-defined minimum time span. This action allows a server restart with less impact on clients than an immediate restart. The latter can occur while some threads may be actively performing work for clients. However, waiting with a restart until the target is idle is appropriate only if a detected problem is not severe enough to require immediate handling and if all threads move to an idle state within an acceptable time after the problem occurrence.

JAS checks whether threads or the entire target is hung in time intervals specified by the user. JAS considers a thread or the target hung if it has not terminated within a user-defined maximum execution time. At the time of the check, JAS transmits the thread and application status (hung or not) to the frontend. JAS broadcasts every detected event to the frontend provided that the user did not direct JAS to ignore the event.

The JAS frontend, shown in Figure 1, visualizes events and actions as well as certain target status data such as the current number of threads. JAS and the frontend will attempt to re-establish the communication link between themselves if it gets interrupted due to a failure of the communication subsystem. The frontend also logs every problem report that JAS sends. The event log allows a user to pinpoint the nature of the detected problem and the time in milliseconds and location in the application of the problem occurrence. An excerpt from an event log is presented in Figure 2. It was generated by JAS supervising WebCompanion, a prefetching and caching Web proxy described in [7].

4 Configuring JAS For a Target Application

Events and actions to be taken by JAS in response to events are expressed as a sequence of policies in a user-specified configuration file. Depending on the knowledge about application internals, ranging from no knowledge to complete knowledge, a JAS user can tailor the JAS configuration to varying degrees. The configuration manager, which is part of JAS, generates a default configuration from the target bytecode. The default configuration can be modified by the JAS user. To keep JAS and JAS configurations simple and to reduce the execution time overhead that JAS imposes on the target application, there is only a fixed set of events and actions that the user can choose from when specifying policies. Currently, policies cannot be changed at run-time.

A JAS configuration is an ASCII file containing policy specifications and other information for JAS. A policy describes what action(s) to take if a specified event occurs. Each policy contains an event type and usually two actions. The first action is triggered by an event of that type as long as the total number of events of that type does not exceed a certain maximum within a certain time window (probation). The maximum and the probation specifications are also part of the policy. The second action is triggered if the specified maximum number of events of that type during the probation has been exceeded.

A JAS configuration consists of a set of thread policies, a set of system policies, and a manager specification. Figures 3 and 4 show excerpts from a sample configuration for the target application WebCompanion. Comments can be freely interspersed in the configuration file, and Figures 3 and 4 show the syntax of the configuration file in the form of comments.

The expressive power of JAS policies is limited and the syntax is unsophisticated. On the
Figure 1: Graphical frontend for JAS

Figure 2: Excerpt from a JAS event log showing the times when events occurred (in milliseconds after program start), the events, and actions taken

other hand, it is very easy to learn the policies syntax and to write JAS policies. Also, JAS can parse policies specifications very quickly. This speeds up application restarts (JAS processes the configuration file during each restart). For these reasons, we decided against designing a flexible and powerful, yet easy-to-parse policies language and forcing the user to learn it. Instead, we are
PrefetchThread

// abnormalThreadDeath <maximum> <probation> <beforeAction> <afterAction>
abnormalThreadDeath 2 3600000 notify restartIdleApp

// naturalThreadDeath <maximum> <probation> <beforeAction> <afterAction>
naturalThreadDeath 0 INFINITE none quit

// expectedCompletionTime <time> <maximum> <probation> <beforeAction> <afterAction>
expectedCompletionTime INFINITE INFINITE INFINITE INFINITE none none

// softLimit <number> <maximum> <probation> <beforeAction> <afterAction>
softLimit 100 INFINITE 300000 none restartIdleApp

// hardLimit <number> <maximum> <probation> <beforeAction> <afterAction>
hardLimit 200 INFINITE 300000 none restartIdleApp

Figure 3: Policies for thread class PrefetchThread in the JAS configuration for Web Companion

working on a Java interface that allows users to implement their own policies in Java in the next version of JAS. This interface is also mentioned in Section 7 and it can be used whenever users feel that the restricted policies in the JAS configuration file are not expressive enough.

4.1 Thread Policy Specifications

Every thread in Java is generated from an object that is of class java.lang.Thread or a subclass thereof. In other words, every thread in Java can be naturally associated with a class that defines the behavior of the thread. For each such class, the JAS user may add a set of policies to the JAS configuration. These policies determine what performance and reliability-related events originating at a thread of the specified class JAS ought to consider a problem and how to respond to them. JAS does not require a class used in a policy to be publicly accessible or be present when JAS starts up. Policies can thus describe private inner classes and classes that the target application loads dynamically at a later time. Currently, JAS allows the specification of five policies for each thread class. Figure 3 shows policies for the thread class PrefetchThread in Web Companion. The five policies for each thread class concern the following events:

1. abnormal thread termination (caused by an uncaught exception)
2. normal thread termination (run method comes to a natural end)
3. expected completion time for thread has been exceeded (thread hung)
4. soft limit for number of threads has been exceeded
5. hard limit for number of thread has been exceeded

The soft limit and the hard limit are simply two different threshold values for the number of threads, where the hard limit is at least as large as the soft limit. The idea is that the user may want different policies to apply when different threshold values for the number of threads have been reached. For simplicity, only two different threshold values can be specified. When the lower of the two threshold values (soft limit) has been reached, a less severe action may be required from JAS than when the higher of the two threshold values (hard limit) has been reached. Reaching a soft or hard limit on the number of threads can imply resource or performance penalties that the user would like to avoid. It can also mean that there is a bug in the program that causes more than an allowed number of threads to be spawned. In the former case, the specified action could be, for example, suspend, which means that each newly spawned thread that exceeds the given threshold is suspended immediately until the total number of active threads has fallen below the threshold. At this time, JAS will start the execution of the previously suspended thread.

To illustrate the specification of thread class policies, consider abnormalThreadDeath 2 3600000 notify restartIdleApp. It means “if 1 or 2 threads of the given class terminate abnormally within
System

// maxRestarts <number>
maxRestarts 3
// systemExit <except> <action>
systemExit 0 restartApp
// VMSHutdown <action>
VMSHutdown quit
// supervisorInternalError <action>
supervisorInternalError quit
// expectedCompletionTime <time> <afterAction>
expectedCompletionTime INFINITE none
// softLimit <number> <maximum> <probability> <beforeAction> <afterAction>
softLimit INFINITE INFINITE INFINITE none none
// hardLimit <number> <maximum> <probability> <beforeAction> <afterAction>
hardLimit INFINITE INFINITE INFINITE none none
// GCmaxTime <time> <maximum> <probability> <beforeAction> <afterAction>
GCmaxTime 500 5 60000 notify restartIdleApp
// GCmaxFrequency <time> <maximum> <probability> <beforeAction> <afterAction>
GCmaxFrequency 150 10 60000 notify restartApp
// checkInterval <time>
checkInterval 3000
// monitorAllExceptions yes | no
monitorAllExceptions yes
// catchThrowableAllocations
{
//  [(Throwable subclass) <maximum> <probability> <beforeAction> <afterAction>]*
// }
catchThrowableAllocations
{
    java.lang.ArrayIndexOutOfBoundsException INFINITE 60000 restartApp restartApp
    java.lang.OutOfMemoryError 3 INFINITE notify restartIdleApp
}

Manager

// hostName <hostName>
hostName manager
// portNumber <portNumber>
portNumber 3333
// timeout <time>
timeout 3000
// maxAttempts <number>
maxAttempts 25

Figure 4: System policies and manager specification in the JAS configuration for Web Companion

any window of 3600000 milliseconds, notify the manager; if more than 2 threads of the given class terminate abnormally, restart the entire application once all threads are idle” (idleness of threads is explained in Section 3). Another example of a policy is naturalThreadDeath 1 INFINITE quit quit. It means “if a thread (i.e., more than zero threads) of the given class terminates normally from the virtual machine point of view, terminate (quit) the application immediately”, implying that either this thread is supposed to run forever but a bug might lead to thread termination, or the death of this thread marks the normal end of the target application execution.
4.2 System Policy Specifications

System policies affect the entire target application and not just individual threads or objects. As the example in Figure 4 shows, JAS currently prescribes ten system policies regarding the following events:

1. number of restarts triggered by JAS in response to detected problems has reached limit; JAS then shuts down the application
2. target application has called System.exit(n), where n is different from user-supplied parameter, indicating a fatal error from which it cannot recover; this event also triggers a virtual machine shutdown
3. virtual machine shutdown, either because the application has terminated or the virtual machine has encountered an internal fatal error
4. JAS has encountered a fatal internal error that it cannot recover from
5. target application has not terminated within the expected time (application hung)
6. entire number of running threads in the application has exceeded soft limit
7. entire number of running threads in the application has exceeded hard limit
8. contiguous garbage collection activity took longer than specified
9. time between two subsequent garbage collector runs is below minimum time threshold
10. an exception object (an object of class Throwable or one of its subclasses) was allocated or an exception handler called a method on an exception object (i.e., an exception was caught by the application); the user may specify a different policy for each exception class

The checkInterval is the time span between subsequent checks for thread hangs and an application hang. If monitorAllExceptions is set to yes, JAS will report and log all exceptions, whether they are handled by the application or not and regardless of the exception class.

4.3 Manager Specification

The manager specification (see Figure 4) describes on which host the frontend resides and under which TCP/IP port number. Furthermore, a TCP/IP socket operation timeout can be specified as well as the maximum number of reconnection attempts in case of a communication link breakdown between JAS and the frontend. If, after this maximum number of reconnection attempts, the connection could not be re-established, JAS will cease to broadcast information to the manager from that time on. The manager specification may be omitted from the configuration, in which case JAS does not visualize or log events and actions.

5 JAS Architecture and Implementation

Because no current Java virtual machine has been designed with built-in application supervision mechanisms, the design and efficient implementation of JAS posed a major challenge. Furthermore, this also limits the range of detectable reliability and performance problems as well as the range of responses to detected problems. The JAS architecture is depicted in Figure 5. JAS is comprised of a shell script, a supervisor agent, a set of modified Java API class implementations, a generic application wrapper, a configuration manager, and a frontend that resides on the same or a different machine and communicates with the supervisor agent via TCP/IP. Except for the (optionally running) configuration manager and the frontend, all JAS components are part of the target application process. JAS is currently implemented for Microsoft Windows NT and Sun Solaris 2.6. It requires Sun's Java virtual machine 1.2 or higher.

The shell script starts up and shuts down JAS and the target application and supplies the appropriate parameters to JAS and the Java virtual machine. To execute JAS, the user types the
name of the shell script `Supervise` followed by the name of the target application and possibly target application parameters. The shell script is also responsible for limiting the number of target restarts to the maximum specified in the JAS configuration. To this end, the shell script receives a return value from the target when the latter terminates. This return value is the number of restarts remaining. Once this value has reached zero, the shell script will not execute the target again.

The tasks of the application wrapper are:

- reading in and parsing the JAS configuration
- transmitting policies from the configuration file to the supervisor agent
- loading the target application via the Java reflection mechanism
- starting the target application
- starting the periodic check for hung threads and a hung target application

The application wrapper, implemented in Java, passes each policy from the configuration file through calls to native methods to the supervisor agent which is implemented in C++. The supervisor agent stores these policies in a hashtable for fast access during target execution. Once the application wrapper has finished its job, it essentially disappears and leaves the work to the supervisor agent and the Java API class modifications.

The supervisor agent is the core of JAS. It is a shared library and attaches to the Java virtual machine through the Java Virtual Machine Profiler Interface (JVMPi) and the Java Native Interface (JNI) [6]. The supervisor agent is written in C++. We chose C++ over Java as the implementation language for the supervisor agent because the JVMPI and JNI are C/C++ interfaces and because maximum speed for the supervisor agent is a prerequisite for low overhead for JAS. Another reason for implementing the supervisor agent in C++ is the fact that many JVMPI event notifications are delivered with the garbage collector turned off. If the virtual machine attempts to execute Java code during a time when the garbage collector is turned off, the virtual machine might lock up.

The supervisor agent first stores all the policies coming from the generic application wrapper in a policies hashtable. It then requests notifications from the JVMPI of events that the policies describe such as

- thread start (necessary for checking soft and hard limits)
- thread termination
- garbage collection start and end
- virtual machine startup completed (necessary for JAS itself)
- virtual machine shutdown

The JVMPI does not deliver the following events to JAS:

- abnormal thread termination
- `System.exit(n)` invocation
- throwing/catching of an exception

These events are communicated to JAS by API classes whose implementations we modified slightly. The thread start event also receives assistance from a modified API class implementation. The classes that we modified are all part of the `java.lang` package and are `Class`, `ClassLoader`, `Thread`, `ThreadGroup`, `Runtime`, and `Runnable`. The implementation changes are not visible to users of those classes and are very minor, essentially consisting of a few new lines of code in certain methods. Porting these changes to a new Java JDK therefore requires little effort.

The modified `ClassLoader` and `Class` implementations assign a unique integer identification number (class ID) to each subclass of `Runnable` and `Thread` that appears in the JAS configuration. This includes inner classes and dynamically loaded classes. Subclasses of `Runnable` and `Thread` pass the class ID to the supervisor agent during certain method invocations that indicate JAS-relevant events. The `Runnable` implementation amends all methods by a new line of code that informs the supervisor agent of the method invocation. This indicates to the supervisor that the application has thrown or caught an exception. The `Thread` class implementation makes the class ID a part of the thread name but hides that fact from a `Thread` class user by only returning the original thread name without the class ID during a `Thread.getName()` method invocation. Because the class ID is part of the thread name, JVMPI events that retrieve a thread name, such as a thread start, pass the class ID automatically to the supervisor agent. The supervisor uses class IDs as keys into a hashtable that stores class policies as defined by the user in the JAS configuration. Such policies determine which action to take in response to the detected event. In the case of a frontend notification, the supervisor agent sends a problem report in string format to the frontend. After the expiration of all probation periods for an event defined as a problem by some policy (e.g., abnormal thread death), the supervisor agent sends another message to the frontend. The default frontend uses this notification to reset the visual indication of a problem.

The uncaught `Exception` method in class `ThreadGroup` has been amended to notify the supervisor of an abnormal thread death. The supervisor receives the class ID and a stack trace from the modified uncaught `Exception` method. The stack trace is eventually sent to the JAS frontend by the supervisor. The `Runtime.exit` method sends a signal to the supervisor including the exit code that the target passed to a `Runtime.exit` or `System.exit` call (which in turn calls `Runtime.exit`). Depending on the JAS configuration, JAS shuts down or restarts the target.

If a thread start event occurs and the supervisor agent determines that the spawned thread would exceed a soft or hard limit and that the appropriate action for this event is thread suspension, the supervisor agent puts the execution of this thread on hold. This is done through a JVMPI method call for thread suspension. Whenever a thread termination event occurs, the supervisor agent checks whether it is safe according to the configuration to resume the execution of any suspended threads and if it is, resumes the execution of the thread that has been suspended for the longest time. Thread execution resumption is effected by a JVMPI method call.

The supervisor agent also periodically executes a thread and application check that determines whether any thread or the entire application is hung. This is accomplished by a separate Java `check thread` with maximum priority that wakes up in user-defined time intervals. The supervisor
records the start time of each thread and of the application in a thread or application control block, respectively. JAS enters all control blocks in a linear list. The check thread scans the list whenever it wakes up. For each control block, it compares the current time with the thread or application start time, respectively, and determines whether the elapsed time exceeds a configuration-dependent limit for the thread or for the application. Every time the check thread wakes up, it also sends a signal to the frontend indicating that the target process is still alive. If the frontend does not receive such a signal within a certain time interval, it may consider either the communication link to JAS broken or JAS to have silently failed. If the communication link is the problem, JAS will notice this and attempt to re-establish the connection to the frontend repeatedly until it either succeeds or considers the communication link to have failed permanently. In a future release of JAS, if no connection request from JAS is received by the frontend within the specified time interval, it may decide to start a copy of JAS and the target application on another machine. This amounts to an automatic failover capability of JAS.

JAS executes a restartThread action following the death of a thread of a user-defined class by spawning a thread of a JAS-implemented class StarterThread. Let us call the user-defined class T. The run method of class StarterThread calls the run method of class T and thus re-executes the T thread that just died. The member variables of T retain the values that they had when the T thread last terminated. In general, this may lead to unexpected results and thus restartThread should be used with caution in the JAS configuration. However, there are many cases where this type of thread re-execution has the desired effect of recovering from a transient fault in a thread without leaving the application in an inconsistent or unexpected state. JAS ensures that the newly spawned StarterThread is subject to the same JAS policies as the T thread. It does so by passing the same JAS class ID to the supervisor agent when the StarterThread encounters a problem as it would when a T thread encounters a problem. From the supervisor agent’s point of view, therefore, the StarterThread masquerades as a T thread.

Another action that JAS can execute and that merits further explanation is restart idle application. When the JAS configuration calls for this action in response to an event, JAS sets a boolean flag that is read by the aforementioned check thread whenever it wakes up. For each thread in the target application, the check thread in JAS determines via the JVMPI ThreadHasRun method whether the thread has consumed CPU time during the time interval between the last execution of the check thread and the current one. If none of the target application threads has consumed any CPU time, JAS restarts the application. By adjusting the check thread time intervals in the JAS configuration, the user can determine how long all threads must remain idle before JAS considers the target application idle.

6 JAS Performance

This section quantifies and exemplifies our claims that JAS can be used to enhance the availability of long-running applications and that JAS is lightweight as far as performance and memory overhead is concerned. The JAS performance overhead depends on several factors apart from the obvious such as Java virtual machine and hardware speed:

- number of thread classes specified in the JAS configuration
- number of exception classes specified in the JAS configuration
- configuration-dependent frequency of JAS thread check (see Section 5)
- number of concurrently executing threads in target application
- total number of JAS-relevant events generated by the target application
- frequency with which JAS sends notifications to the manager

The memory overhead imposed by JAS depends on the first four items listed above.
Subsection 6.1 describes an experiment measuring the effectiveness of JAS in enhancing long-term availability of a server. It also describes two actual uses of JAS at Bell Labs. For all other experiments presented in Subsections 6.2 and 6.3, we let JAS supervise five different applications, each one representing a different type of application and stressing different features of JAS. Three of the applications are synthetic applications which we expected to incur high memory and performance overheads. The first one, synth1, sets and retrieves the name of a thread, a rarely used but time-intensive operation for JAS, one million times. The second synthetic target application, synth2, throws an exception one million times. Since JAS monitors exceptions, we expected this to be time-consuming for JAS to handle. The third synthetic application, synth3, is a long-running computing-intensive application that spawns a total of more than twenty million threads. It does so by creating 20 to 200 threads of \( n \) different classes simultaneously, in increments of 20 threads, and repeats this process for \( n \) from 1 to 20. Each time \( m \) threads of \( n \) classes are created, this procedure is repeated 1000 times. This program shows how the JAS performance overhead scales with the number of simultaneous threads and the number of different thread classes. Another test application is Web Companion, the prefetching and caching Web proxy mentioned in Section 3, which has a fixed set of threads and is computing- and I/O-intensive. The second non-synthetic test applications is version 2.02 of Jigsaw, a Web proxy and server from the World Wide Web Consortium [15]. Jigsaw is largely I/O-bound and has a static thread set in our experiments. In the Web Companion and Jigsaw performance experiments in Subsections 6.2 and 6.3, WebWatch, an automated browser acting as a workload generator and measurement tool\(^1\), simulated one client and requested about 15000 locally stored HTML pages, from Web Companion and Jigsaw, respectively. The test platform was a 400 MHz Intel Pentium II computer with 128 MB of main memory running Windows NT 4.0 and JavaSoft’s JDK 1.2.1 with the JavaSoft Classic Virtual Machine and just-in-time compilation turned on.

6.1 Experiments and Experiences with JAS

6.1.1 Jigsaw

In order to quantify some of the benefits of JAS, we ran an experiment with Jigsaw. This experiment capitalizes on our discovery of a reliability problem when Jigsaw is configured as a proxy. After a large number of requests from a large number of concurrent clients, Jigsaw clients frequently experience problems in retrieving documents. These problems are failed connection attempts, connection breakdowns, responses indicating HTTP errors, and empty responses. There are no other visible signs of any Jigsaw problems such as the display of Java exception stack traces on the console that started Jigsaw.

We set up a JAS configuration for Jigsaw by running the JAS configuration manager on the Jigsaw class file distribution. This procedure does not require manual intervention and therefore takes very little time. The configuration manager creates a human-readable ASCII file. We set one parameter in this configuration file instructing JAS to report all exceptions during the Jigsaw execution to the graphical JAS frontend. We then exposed Jigsaw to a heavy request load from many concurrent clients. We found that SocketExceptions coincided with the beginning of large numbers of failed client requests. Since Jigsaw does not report these exceptions on the console, it must catch the SocketExceptions, perhaps in a not completely successful attempt to recover from these exceptions.

We then reconfigured JAS to respond to SocketExceptions with a restart of the proxy. Reconfiguration meant adding the line `java.net.SocketException 2 INFINITE notify restartApp to the catchThrowableAllocations section (cf. Figure 4) in the JAS configuration file. This pol-

\(^1\)WebWatch is a program developed at Bell Laboratories. More details about WebWatch can be found in [7].
icy means that JAS will report the first two SocketExceptions to the frontend. If there is a third SocketException, JAS will restart the proxy. The INFINITE keyword tells JAS to never reset the count of SocketExceptions, i.e., JAS will restart the proxy when three SocketExceptions have occurred throughout the lifetime of the proxy. This policy reflects our feeling that one SocketException does not have a disastrous effect on the proxy performance but that the proxy performance quickly degrades with further SocketExceptions.

After the JAS reconfiguration, we conducted an experiment in which WebWatch simulated 75 concurrent users that retrieve a total of more than 15000 HTML documents from about 100 actual Web sites through the Jigsaw proxy. The experiment was performed once with Jigsaw under JAS supervision and once without JAS supervision. In each case, the experiment consisted of three runs, and we averaged the outcomes over the three runs. Without JAS, WebWatch successfully retrieved the requested HTML document in only 60.64% of all cases. With JAS, the number of successful HTTP requests increased dramatically to 93.73%. Moreover, the average time for successfully processing a client request was 21.56% higher when Jigsaw was run without JAS. Apparently, once Jigsaw gets into an unstable state after the first few SocketExceptions, it takes Jigsaw longer to process client requests. Figure 6 shows how Jigsaw performed under the supervision of JAS and how without JAS. The dotted line, representing the percentage of successful HTTP requests among all HTTP requests to Jigsaw under the supervision of JAS, has several dips to 0% but otherwise remains close to 100%. The dips to 0% happen during automatic restarts of Jigsaw initiated by JAS in response to SocketExceptions in Jigsaw. After each restart, the reliability of Jigsaw returns to 100% for a while until the next exception occurs. The solid line, representing successful HTTP requests among all HTTP requests to Jigsaw without JAS supervision, clearly shows a sudden decrease in the percentage of successful HTTP requests due to an exception in Jigsaw. From this point on, the percentage of successful HTTP requests fluctuates between 0% and about 92%.

Figure 6: Percentage of successful HTTP requests to Jigsaw with and without JAS supervision
6.1.2 JAS Use At Bell Labs

We currently use JAS to supervise the execution of two multithreaded Java servers in a call center prototype at Bell Labs. One of the Java servers frequently throws uncaught exceptions towards the end of the lifetime of client request handler threads. This server was developed by a different team at Lucent Technologies and we therefore have no insight into the architecture and implementation of it. At first, the exceptions that the server throws do not seem to affect the functionality or performance of the server. However, after a certain number of client requests to this server it seems to lock up completely. We therefore instructed JAS to preventively restart this server whenever the number of uncaught exceptions has reached a certain threshold. This resulted in high availability of the server except for short time intervals during which the server is restarted by JAS.

The second server that JAS supervises has not exhibited any failures to date. However, we wanted to determine how JAS would handle a failure in a critical component of this server. To this end, we injected a bug into a central dispatcher thread. This thread receives login notifications and data from call center agents. If it fails, the entire server is paralyzed. Our artificial bug gets triggered when an agent with an illegal login name (i.e., a name containing a non-alphanumeric symbol) attempts to log into the server. The bug throws an uncaught IllegalArgumentException that leads to the death of the dispatcher thread. We configured JAS to restart the dispatcher thread whenever it terminates. Under JAS supervision, the bug prevents an agent with an illegal login name from logging into the server. Other agents who simultaneously transmit data to the server also experience a server failure. However, after the almost instantaneous thread restart, all subsequent agent logins and data transmissions to the server proceed as expected. Just as described in the previous two examples of servers under JAS supervision, a (partial) server execution failure leads to a brief server outage but is quickly detected and resolved by JAS resulting in long-term server availability.

6.2 Memory and Storage Overhead

Memory overhead was measured with the five test programs listed above. The results are summarized in Table 1. It shows that very little extra physical memory is required for supervising any of these applications with JAS (see peak virtual byte size). Using JAS, however, noticeably increases the amount of memory that is accessed frequently (see peak working set size).

<table>
<thead>
<tr>
<th></th>
<th>Peak Working Set Size Overhead</th>
<th>Peak Virtual Byte Size Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>synth1</td>
<td>13.15%</td>
<td>2.16%</td>
</tr>
<tr>
<td>synth2</td>
<td>17.29%</td>
<td>2.16%</td>
</tr>
<tr>
<td>synth3</td>
<td>8.68%</td>
<td>1.00%</td>
</tr>
<tr>
<td>WebCompanion</td>
<td>16.43%</td>
<td>1.62%</td>
</tr>
<tr>
<td>Jigsaw</td>
<td>11.68%</td>
<td>0.98%</td>
</tr>
</tbody>
</table>

Table 1: Memory overhead incurred by JAS for five target applications

6.3 Execution Time Overhead

Execution time overhead, measured as the ratio between the total execution time of a program under JAS supervision and the total execution time of that program without JAS supervision, is shown in Table 2. The numbers are averaged over three repetitions of the same experiment and they include optional notifications sent to the JAS frontend. They exclude the time for starting
up the target application where JAS incurred an additional time overhead of less than 1.2 seconds regardless of the target. The third column in Table 2 shows the minimum and maximum observed execution time overheads for the different experiments. We see that the overheads are quite uniform across different runs of the same experiment. As expected, JAS incurs a high execution time overhead for running synth1. The high overhead is due to additional operations that JAS performs to incorporate class IDs into thread names and to hide this from Thread class users (see Section 5). The entry for synth2 actually indicates that running this program under the supervision of JAS results in a speed-up, an unexpected effect for which we do not yet have an explanation. As we had expected, synth3 demonstrated that JAS scales very well to hundreds of simultaneously executing threads. More specifically, the increasing number of threads and thread classes over the course of the execution of synth3 did not result in a corresponding increase in execution time overhead. The WebCompanion and Jigsaw experiments measured the execution time overhead from the client (WebWatch) perspective. Except for synth1, which represents a highly unlikely target application scenario, the measured overheads are well below 10%.

<table>
<thead>
<tr>
<th>Application</th>
<th>Average Execution Time Overhead</th>
<th>Minimum/Maximum Execution Time Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>synth1</td>
<td>65.05%</td>
<td>59.66/69.06%</td>
</tr>
<tr>
<td>synth2</td>
<td>-19.43%</td>
<td>-19.72/-19.2%</td>
</tr>
<tr>
<td>synth3</td>
<td>6.05%</td>
<td>5.71/6.66%</td>
</tr>
<tr>
<td>WebCompanion</td>
<td>2.44%</td>
<td>-3.5/6.15%</td>
</tr>
<tr>
<td>Jigsaw</td>
<td>6.00%</td>
<td>2.57/11.78%</td>
</tr>
</tbody>
</table>

Table 2: JAS execution time overheads for five different target applications

6.4 Restart Time

For servers that require high availability, the time between the occurrence of a problem that requires a restart and the completion of a server restart (restart time) is an important metric. A fast restart after a detected problem amounts to a brief interruption of the services rendered and usually a subsequent clearing of the problem that led to the restart. We demonstrate below that JAS restart times are very short. Thus, JAS can turn a target application that would otherwise suffer from intermittent reliability or performance problems into an application that offers consistently high availability and performance. A similar statement applies to thread restarts. Since they happen virtually instantaneously, we did not measure the time that it takes JAS to restart a thread.

JAS detects problems practically immediately after they have occurred, with the exception of hung threads and a hung application. In the latter cases, the detection depends on how long the user-defined thread/application hang check interval is. The time for the completion of a restart largely depends on the size of the configuration file, the number of classes that the virtual machine has to load upon startup, and the initialization of threads and objects in the application, as deemed part of the completion of a restart. In our test applications, we triggered restarts by purposely throwing a NullPointerException in the first statement of the main() method of the corresponding application. The application was restarted 50 times, and the restart time was averaged over the 50 restarts. This experiment was repeated three times for each application, and the final restart time per application was computed as the average of the outcome of the three experiments. The results are shown in Table 3. We may safely assume that after each restart, the Java virtual machine loads class files from computer memory and therefore the results in Table 3 do not include the times for loading classes from disk.
<table>
<thead>
<tr>
<th>Application</th>
<th>Average Restart Time (ms)</th>
<th>Minimum/Maximum Restart Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>synth1</td>
<td>821.73</td>
<td>815.2/834.2</td>
</tr>
<tr>
<td>synth2</td>
<td>825.6</td>
<td>820.4/835.2</td>
</tr>
<tr>
<td>synth3</td>
<td>927.27</td>
<td>921.8/936.6</td>
</tr>
<tr>
<td>WebCompanion</td>
<td>837.8</td>
<td>838/838.8</td>
</tr>
<tr>
<td>Jigsaw</td>
<td>1191.4</td>
<td>1190.2/1192.8</td>
</tr>
</tbody>
</table>

Table 3: Average restart times for five target applications

The table demonstrates that the average restart times are very short and that they are very uniform for a given application. The differences in restart times among applications are due to a different number of classes that need to be loaded at start-up time and due to configuration files of very different lengths. Jigsaw has by far the most classes and the longest configuration file among our test target applications.

7 Work in Progress and Future Work

We are currently working on or planning to work in the future on several extensions to JAS. There will be several new actions in response to detected problems including failover/migration as an alternative to simple program restart or shutdown, and spawn copy for load balancing. Currently, policies in JAS are static and cannot be changed after JAS has started up. One of our goals is to allow dynamic changes to policies while JAS is running. JAS will be able to detect many more performance and reliability problems such as excessive number of objects, high memory or CPU consumption, thread deadlock, illegal method call sequences, etc. We are also planning on integrating JAS into other frameworks, e.g., into cluster managers, network management networks, and fault-tolerant CORBA implementations [12]. Currently, we are working on a Java interface for JAS that allows the user to program highly customized event handlers that can be plugged into JAS and allow more flexibility than the simple policies in the JAS configuration. These event handlers will be called by JAS when certain events happen and will be supplied with relevant parameters that JAS keeps track of. The event handlers translate events and JAS parameters into actions including JAS actions listed in Section 3.

References


REFERENCES


