# WRITING TINE SERVERS IN JAVA

Philip Duval and Josef Wilgen
Deutsches Elektronen Synchrotron /MST

*Abstract*

The TINE Control System [1] is used to some degree in all accelerator facilities at DESY (Hamburg and Zeuthen) and plays a major role in HERA. It supports a wide variety of platforms, which enables engineers and machine physicists as well as professional programmers to develop and integrate front-end server software into the control system using the operating system and platform of their choice. User applications have largely been written for Windows platforms (often in Visual Basic). In the next generation of accelerators at DESY (PETRA III and VUV-FEL), it is planned to write the TINE user-applications primarily in Java. Java control applications have indeed enjoyed widespread acceptance within the controls community. The next step is then to offer Java as a platform for front-end servers.

In this paper we present the TINE Java server API and first results using TINE Java servers. In particular we shall discuss the pros and cons of using Java as a platform for front-end and middle-layer servers and present timing results concerning Java servers using native Java, Java plus JNI, and native TINE servers.

## INTRODUCTION

The TINE Control System is a distributed, object-based system which runs on most platforms (legacy as well as modern) offers numerous services (both central and distributed) and enjoys a widespread use in the accelerator facilities at DESY. The TINE protocol offers high performance under extreme circumstances, such as transmitting large video frames at many hertz to multiple clients via multicast. Historically, no matter what platform or what language is being used for TINE clients or servers, the core TINE kernel has been written in C. Thus clients and server using say Visual Basic in WINDOWS make use of ActiveX controls (or VBX controls) or direct DLL calls which themselves call into the TINE kernel. Likewise, clients or servers written in LabView make use of VIs which interface to either TINE DLLs or shared TINE libraries, and so on. The Java TINE client API on the other hand interfaces to tine.jar, which is written entirely in Java. Certain aspects of the TINE C kernel were ported to Java, but to a larger extent, the TINE kernel was simply rewritten in Java, taking full advantage of the Java language where possible. All efforts were made to maintain as many similarities in the basic TINE API as possible. This does not mean that they are identical. Calls such as "AttachLink()" and "ExecLink()" are represented by the methods "attach()" and "execute()" of the representative TLink Object. Note that the TINE protocol does not deal so much with 'puts' and 'gets' as with data 'links'.

One's first inclination when offering Java in the Control System's portfolio is to say that we don't need to worry about a Java Server API since front-end servers will always have to access their hardware and that is best left to code written in C. Furthermore, if there are real-time requirements, Java would not be an acceptable platform owing to Java's garbage collection kicking in at indeterminate intervals.

Nevertheless, Java is a powerful language and offers numerous features and a wonderful framework for avoiding and catching nagging program errors. Thus there does in fact exist a strong desire to develop control system servers using Java. If these are to be middle layer servers, which manage and interpret data from front end servers, then the hardware issue is moot. Even front end servers can be written in Java when the hardware IO is made available by other means, such as a JNI or a CNI interface to the C libraries which do the 'dirty work'.

It still remains to clarify whether issues of performance or garbage collection preclude writing effective servers in Java.

A first effort has now been made to include the TINE server API within the tine.jar Java archive, so that we can make the initial performance tests. We shall report on these below.

## INITIAL RESULTS

The current Java TINE server prototype offers approximately 75 percent of the functionality of a standard TINE server. Missing are such subsystems as the local alarm system and the local history system. Furthermore the current prototype does not offer initialization via a configuration database. For our purposes at this juncture however, these are trivial points. The server developer will not deal directly with these aspects in any case. The fundamental server management kernel and API are available, and it is primarily these which we present here.

We first compare several Java Virtual Machines (JVMs) with regard to fairly general concerns, not specifically related to a TINE server. Then we shall turn our attention to the performance of our TINE server prototype.

### *Effects of Garbage Collection*

We can get a handle on the side-effects garbage collection might have on our test server application by having a look at the repetitive instantiation of objects. By instantiating objects in a tight loop we notice that there are occasional delays on the order of 10 to 100 milliseconds, depending on the Java Virtual Machine

(JVM) being used, which appear as spikes in a trend chart of the instantiation execution time. It also turns out that not only the delay is dependent on the JVM, but the frequency intervals with which these delay spikes appear depends on the JVM being tested. At the programmer level, such spikes can sometimes be avoided by using available programming techniques (such as Object caching) and exercising a certain discipline concerning 'new' operations. However this is not always possible or desirable and sometimes defeats the purpose of using Java in the first place. On the other hand, in normal operation a server will not be instantiating objects to such a degree and will be more or less operating in a steady state. To be sure, as clients come and go, the TINE connection tables will be populated or de-populated. Furthermore, Java methods such as 'toByteArray()' will in fact inherently create objects which are destined to be discarded. As long as this is managed efficiently, the effects of garbage collection can be minimized. The acid test will in fact involve running a TINE Java device server under realistic conditions.

A trend chart showing the magnitude of such delay spikes is shown below in figure 1. Possible effects of adjusting configuration parameters in the Java Virtual Machines were not investigated.

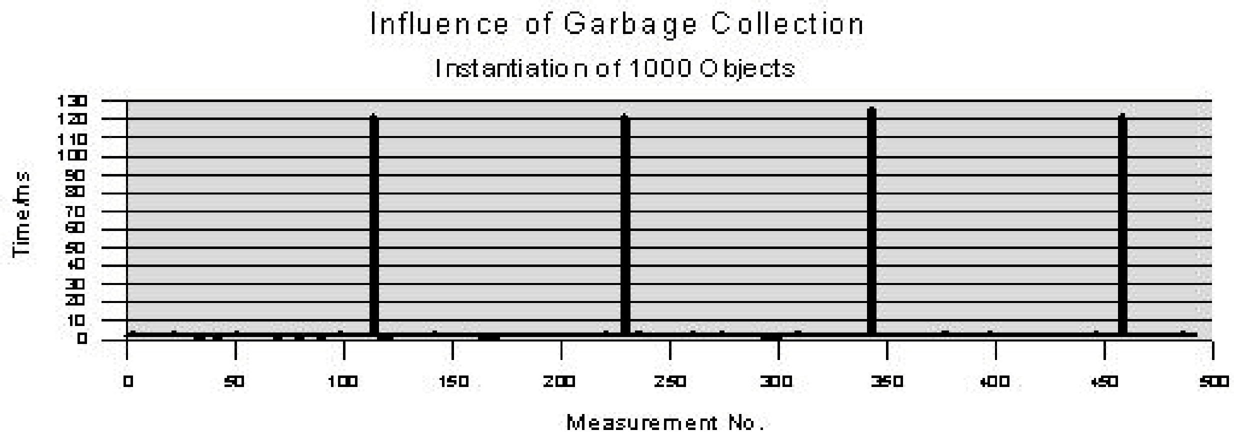


Figure 1: The influence of garbage collecion on the instantiation of objects in a Java Virtual Machine.

## *JVM Performance Comparison*

We have examined certain performance characteristics for the following Java Virtual Machines:

- J2SE 1.4 und 1.5
- j2me, Personal Edition
- kaffe
- jamvm, sablevm
- gcj, gij

In making benchmark comparisons we focused on the following aspects:

- A comparison of the performance characteristics and required resources of byte-code interpreters, JIT compilers, and at least one native compiler. In particular, how do the alternative JVMs stand up in comparison to J2SE/J2ME?
- Is Java a viable solution on systems with limited resources?
- What can we say about the performance of JNI (Java Native Interface) regarding 1) its usage on different JVMs, 2) the different methods of accessing objects, and 3) a comparison with CNI (Cygnus Native Interface)?

If we lump assorted benchmark tests (such as object instantiation, matrix multiplication, hash list access, File IO, Exception handling, etc.) together, we can get an idea of an overall comparison by looking at Figure 2. below. In this comparison we see that J2SE along with Kaffe and GCJ all outshine the alternative JVMs J2ME, jamvm, sablevm, and GIJ. As J2ME, jamvm, sablevm are pure byte-code interpreters, this is not surprising.

On the other hand, if we focus on the resources needed by the JVM, we note that J2SE 1.4 and 1.5 demand considerably more disk space than any of their rivals.

The time required for loading the JVM and starting a Java server was seen to be fairly uniform across the JVMs examined with the exception of J2ME, which was a factor of 6 faster than, for example, J2SE 1.5.

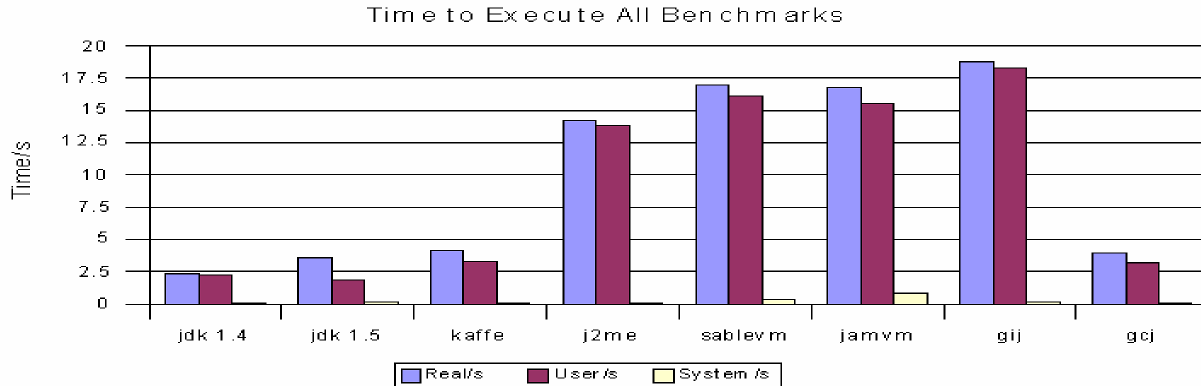


Figure 2: Rough comparison of general benchmark execution time for different Java Virtual Machines.

As to a comparison of the respective performances relating to access to C libraries via JNI, we note that most JVMS performed more or less equally well regarding getting and setting fields. The marginal, overall "winner" was Kaffe, largely due to its handling of method calls (J2SE 1.5 was a factor of 3 slower). In a category all it own was GCJ, which alone makes use of the CNI interface. As GCJ compiles the java code to native, the interface to C libraries (after the fact, as it were) is expected to more efficient and it is. In fact, with CNI and GCJ the overhead of all manner of access (get/set fields, method calls) is essentially negligible. Furthermore, the interface to a C library is in this case straightforward and requires no java 'stub' as in the case of JNI.

## *TINE Server performance*

The bottom line is of course: Will my Java server run stably and steadily, unencumbered by problems of resource depletion or garbage collection, etc? For a wide variety of device-servers common to accelerator controls this is surely true. The benchmarks alluded to above were made on a "run-of-the-mill" 500 MHz Pentium III PC with 256 Mbytes of RAM. Even on such a machine, the worst performing JVM above would be adequate for delivering "slow control" parameters to a number of clients at 1 Hz.

To help quantify these assertions, we examine a TINE java server, which does nothing more than update a sine curve at 10 Hz (by itself somewhat computation intensive). The server also offers properties to get and set the frequency and amplitude of the sine curve. We are interested in the reliability of data acquisition from a number of clients being updated at 10 Hz, and in the reliability and turnaround-time of accessing one of the properties as a 'get' call (for instance, issuing a get call inside a tight for loop) . If both tests are made simultaneously, it should also show any effects of garbage collection which might arise on a device-server running in steadystate. If the client making the 'get' calls is not the same as one of the clients receiving the sine curve, it will come and go in the server's client table.

The initial steady-state test involves running three clients each requesting a sine curve trace (256 double float values, i.e. circa 2 KByte payload) at 10 Hz. One can easily keep statistics at the client side to determine whether an expected incoming packet "misses" (either fails to come or is outside the 100 millisecond response window). A 24 hour test shows no misses, as long as the server and clients are allowed to run without being influenced by starting and stopping other applications on the same machine. The steady-state can easily be disturbed by starting, say, a web browser or word processor. But this is expected and under normal conditions is not applicable. Servers typically run in a "dark corner" somewhere are generally not being used as web browsers or word processors. The same holds true for client programs running on consoles in the control room.

The second test involves issuing several thousand synchronous 'get' commands inside a loop and examining both the reliability and turn-around-time. We should point out that the current TINE server prototype offers only UDP communication (which is the default TINE protocol), so in order to minimize packet loss, the client and server run on the same subnet and on the same switch segment. The initial results are likewise very encouraging. There were no timeouts or dropped requests (over several hundred thousand attempts), and the turnaround time was seen to be on the order of 2 milliseconds per call for J2SE 1.4. Although this is a factor of 5 or so larger than the turnaround time when accessing a C native server under similar conditions, it is nonetheless acceptable as is, especially when considering that the current TINE Java server is only a prototype which has not yet been fully optimized.

Finally, we run both tests together in order to test reliability of the incoming sine data against possible distortion due to garbage collection where client objects are being constantly created. Repeated testing showed no discernable differences in performance as long as the clients running the independent tests were on different machines.

## CONCLUSIONS

Java is absolutely suitable for device servers which do not have real-time requirements. When large amounts of data are produced, I/O on byte arrays can be a bottleneck, especially with byte-code interpreters.

The native compiler GCJ has almost no performance advantage over a JIT Compiler. The memory consumption is also comparable. The major advantage of using GCJ is the ease in linking C library code via the CNI interface.

A byte-code interpreter can be expected to be a about factor of 10 slower than a JIT compiler. For device servers which do not have CPU-intensive tasks, this should still be fast enough.

Sablevm and jamvm are excellent open source alternatives to J2ME. Both have rich libraries (GNU Classpath) and Sablevm is available for many platforms. J2ME on the other hand has a very limited library which is quite out of date.

It is still too early to say which JVM and which Cinterface will be the preferred solution for TINE servers at DESY. In the end, the ‘preferred’ solution could depend on the platform being used.

## REFERENCES

[1] http://desyntwww.desy.de/tine