Evaluation of Data Acquisition tools based on Java

H. Beker, R. Divià

ALICE collaboration, CERN, Geneva, Switzerland

One of the requirements of today's Data Acquisition systems is a Monitoring and Run Control tool which is easy to use, robust, easy to integrate with the global environment and possibly portable on multiple platforms. The ALICE DAQ group at CERN decided to evaluate the possible use of WWW and Java for development in a quasi-on-line distributed environment. An Event Display utility that was already implemented in C and Tcl/Tk was chosen as a test case and was transposed in Java. We will describe the targets of the project, its evolution and the achieved results.

1 Introduction

Data Acquisition systems conceived for High Energy Physics need Monitoring and Run Control tools whose capabilities should allow easy implementation, quick run-time reconfiguration, robustness and fast integration with external modules and facilities. Nowadays it is considered common practice to do tasks such as run monitor and even run control from desktop computers; this introduces new constraints concerning development tools, run-time environments, portability across platforms, distribution schemes and installation procedures. It also becomes very important to reduce the impact of the data shipping and handling process on the global environment (local and remote CPUs, interconnecting network). Developers have hard choice between public domain and commercial packages. Public domain tools have several advantages over proprietary solutions, but their use in online environment has been quite difficult to accept. On the other hand, commercial products are usually better supported, although their availability and flexibility can be quite poor. Things have recently changed with the introduction of proprietary packages available free of charge whose target is not to sell the product itself but rather to create new markets. These products combine the advantages of public domain and commercial solutions: widespread use, flexibility, availability and extensions. Two examples of such tools are Java and Netscape.

2 Why Java?

Since quite some time it was obvious that WWW-based tools could meet several of the requirements for HEP applications. Unfortunately, a "proper" load balance between server and client hosts was missing and the impact of this type of solution was not acceptable both in terms of workload for the server CPU and for the network. A new approach was needed, such as the one proposed by Java, which combines the distributed features of the WWW with the required load balance between client and server hosts. Pre-compiled code is exchanged between the client (online) and server (desktop) stations and installed at the click of the mouse. A dedicated and optimised TCP/IP link for the exchange of raw data can be easily established. Common libraries are available on all the supported platforms for strings manipulation, math functions, I/O handling and graphical operations. All this gives a proper load balancing, a trivial installation procedure, immediate version coherency guarantee, a lighter impact on the network and a very good availability across different platforms.

For our exercise we took an existing application (Event Monitoring) written in Tcl/Tk and we transposed it in Java. The tool was to be made available on Unix (`Sun Solaris`, `HP-UX`), PCs (`Windows 95`, `Windows NT`) and MACs (`MAC-OS`). The events to monitor - generated by a *Ring Imaging Cherenkov*

(RICH) detector - had a simple variable length record format with an average data size of 6 KBytes. The server host was to publish the hypertext and the Java code using a standard http daemon. It was also providing the raw event data via a dedicated TCP/IP data server. The desktop station could then establish a private data link with the online host to receive the raw data ready to be decoded and plotted. Some extra capabilities had been added to the client side of the application such as data inspection, events navigation and display-on-availability for data-polling loops (run monitor). The data server daemon, written in C, shipped the data via a TCP/IP socket to a Java "Event Link" class; this class was then used for link validation, link evaluation and Event Display.

3 The Event Display

Based on the Event Link class, the Event Display was developed in raw Java code using a standard Unix text editor. It made use of the Abstract Window Toolkit (`java.awt`) available from Sun as part of the Java Application Programming Interface (*API*). Written as an extension of a standard Java Applet, the Event Display could make use of all the libraries available at no cost as part of the "standard" Java Development Kit (*JDK*). The skeleton of the Event Display had three cooperating Threads: the main handler, the commands handler and the link handler. In addition, the link handler had a dedicated Thread to handle timeouts and abort requests. The result was a foolproof scheme where immediate response was always available, even after fatal errors and link hang-ups. The Event link class had to "cooperate" with a tough system module: the socket library. We have been able to use a TCP/IP socket as a common Java I/O input/output streams pair, with some problems due to conversion between different data types (8 bits vs 32 bits entities). Getting data from the link needed some careful tuning. At first we used a byte-by-byte approach but this was not satisfactory (35 μs per byte). We then switched to a "block transfer" mode and the link performance boosted to 2.3 μs/byte (434 KB/s) on non-dedicated machinery and on a public network, including transfer setup and byte swapping. For our exercise we used Netscape on `Unix`, `MAC-OS`, `Windows 95` and `Windows NT` plus Internet Explorer on `Windows 95`. The Applet was almost immediately available on all those platforms. Run-time errors were easy to detect and trace thanks to the Java console available on most of the browsers. Debugging of graphical applications was not trivial and we had to provide our own tracing facilities. Automatic code documentation was immediately available via the *javadoc* tool (distributed as part of *JDK*); using the source code as input, we could produce an exhaustive list of all classes used by the project plus a description of their mutual interactions and put the result directly on the WWW. The whole project took $\sim$ 700 lines of code. Two man-weeks were dedicated to self-training, class development, debugging, evaluation and validation.

4 Conclusions

Used as a "bare tool" Java proved to be a valid alternative to other public-domain and commercial packages. The targets of the evaluation exercise were met, both in functionality and in performance. The object-oriented class-based approach of the Java language made code writing very effective. The communication between the Java code and the system socket library reached performances in excess of 400 KB/s. The result was a tool capable of browsing 2-dimensional data at very reasonable speed ($\sim$ 5 seconds to down-load the Java byte-code, 4/6 Events/second browsing through a static data file). The look-and-feel of the graphical interface was coherent with all the architectures where a Java browser was tried out. Members of the developers' team have been able to run the tool from remote sites (such as computer exhibitions and conferences) with no installation required and reaching acceptable performances.

More information on the Java-based event display (*Jedi*) project can be retrieved from the Jedi home page at the URL "`http://aldwww.cern.ch/Jedi`".