Towards a WebJDK: Extending OpenJDK 7 for client file system access over cloud

C. V. Deshpande, A. S. Shetye, S. S. Ghuge, J. V.Aghav
Department of Computer Engineering and Information Technology,
College of Engineering,
Pune 411-005, India
Email:{deshpandecv10.comp, shetyeas10.comp, ghugess10.comp, jva.comp}@coep.ac.in

***Abstract*—Deploying Java applications to the cloud with access rights to the local client's file-system involves modification of the Java application code to use non-standard APIs for client's file-system access. With the steady migration of existing technologies towards a web-aware implementation, this paper presents the file access sub-system of a potential WebJDK. WebJDK is designed to provide an implementation of the OpenJDK that runs in the cloud and interacts directly with the client's browser. The WebJDK implementation and its sub-systems are based on the Caciocavallo project by OpenJDK. The proposed file access sub-system implementation leverages the File-System API provided by HTML5. This allows the use of standard Java APIs for accessing files of clients with variety of underlying native file systems.**

***Index Terms*—Softwares, Open source software, File sytems, Web services.**

I. Introduction

The recent trend towards deploying applications as cloud based services aims to leverage the availability and scalability benefits afforded by a cloud setup. However significant effort must go into modifying or re-developing existing applications for deployment on cloud. This includes re-engineering existing native applications as web applications, requiring re-development to compatible technologies supported by the web server or cloud platform.

Leading platforms like Google App Engine and Amazon AWS for developing applications for cloud require software developers to develop their aplications with their respective proprietary Software Development Kits (SDK) [1]. In such a situation, migration of existing applications to the cloud requires a great deal of re-development. Specifically for Java applications, most of which were developed as standalone Java applications, today serve important and increasingly complicated roles in organisations. Increasing number of organisations today want to leverage the advantages of cloud computing and re-developing these legacy systems may prove to be very expensive and possibly introduce new bugs. This makes re-development of such applications cumbersome, it is thus better to host existing Java code on a platform that delivers the Java application as a web service hosted on a cloud platform. Approaches for the delivery of Graphical User Interface (GUI) of Java applications over the web by migrating Java-Swing applications to Ajax enabled web-based application are well explored [2]. However client file access still remains an impediment. Often to secure client file access over the web the client needs to have software pre-requisites like Java Web-Start or Java Runtime Environment for file access through applets. This imposes restrictions on the type and variety of devices that can be addressed. Hence a JDK distribution which will serve as a platform targeted towards allowing existing Java code too be seamlessly deployed over cloud is needed. The platform should address the need of client file system access allowing us to open a file in the client's system over the web with the same Java syntax for opening a file locally. This will usher in a truly web-enabled experience not limited to the delivery of the graphical interface only but also client file system access with the benefits of minimal code modification.

II. WebJDK

The WebJDK is a potential OpenJDK based Java distribution built upon the Caciocavallo project [3]. The Caciocavallo project has made possible the porting of Java GUI backend to newer platforms, it is a key component of WebJDK that allows the Swing/AWT GUI to be represented as and interacted with, using pure HTML and JavaScript. The WebJDK [4] is designed to be a web aware distribution which delivers the Java 2-D graphics stack as HTML to the client's browser and provides access to its resources such as the client's file system through Java's standard File system APIs.

This paper addresses the system design and implementation issues associated with developing a file access sub-system for the WebJDK. The paper aims to document the approaches used towards executing file I/O over the network and the related security measures to be employed in the file sub-system for the WebJDK first proposed by its authors [5]. The file access sub-system is designed to be transparent to the programmer, accessed simply by the standard *java.io* class hierarchy. This remote file I/O is achieved by intercepting calls from Java applications to file access methods and redirecting the call to a customised implementation of the java.io class hierarchy. These calls are then processed and forwarded to clients by servelets utilizing the HTML5's FileSystem API [6] to achieve remote file access.

978-1-4673-5758-6/13/$31.00 © 2013 IEEE

III. CACIOCAVALLO FRAMEWORK

Caciocavallo is a framework built to allow porting of Java's graphics stack to newer platforms easily. Based on Caciocavallo, the CacioWeb offering allows the porting of Swing/AWT graphical components into HTML. This includes event handling, windowing and drawing features. The Java application actually runs on the server, however Caciocavallo allows the porting of the Java graphics stack to newer platforms. The solution is somewhat comparable to Virtual Network Computing (VNC) based solutions however the windowing and drawing directives are issued from the Java graphics stack alone unlike VNC [7]. This allows the rendering of a Java application's graphical interface to a client's HTML5 browser. The only client side requirement is that of having an HTML5 capable browser. All execution and graphics processing resides on the server side thus giving us a truly lightweight client. This allows the power of Java applications to be brought to closed platforms like Apple's iOS, iPad etc. Java applications can be easily delivered to clients such as mobile platforms without the need of a JRE or any prerequisites on the client side [8].

The Caciocavallo project achieves the porting by representing heavy AWT components as Swing components. Thus it lets Swing handle all the widget functionalities. Though a programmer may code AWT, the framework runs this as Swing. Thus multiple ports for each AWT widget will not be required. This allows the application to maintain a uniform look and feel and makes the job of porting a little easier. The Java 2D graphics stack as well as its event handling sub-system is implemented by Caciocavallo. The CacioWeb project built on top of Caciocavallo allows the porting of the Java graphics stack to an HTML5 compatible browser. CacioWeb makes it possible to deploy Java applications on servers and deliver their interfaces as HTML to the client.This approach provides several advantages over the newer Java WebStart platform which allows Java to be delivered to client browsers, however requires Java WebStart to be installed on the client's side. This development in GUI portability sets the pace for a WebJDK, much of which has been achieved by the CacioWeb project. Even though Java GUI can be delievered directly to the client's browser, to have a complete WebJDK in place, we must provide access to the client's resources for achieving the illusion that the application is running on the client's device itself. Discussions about such a WebJDK and its features first occur in a release post about Caciocavallo 1.1 [5].

IV. WEBJDK FILE SYSTEM

The WebJDK file system henceforth referred as WebFS, is designed to allow Java application code executed on the server to access the client's local file system. WebFS is a transparent component of the WebJDK that allows programmer to address file I/O operations with the client without having to handle the connectivity and data transfer issues. This means that the Java code deployed on the WebJDK platform will execute in the same was as it would if it were to run on the client's device itself. The use of web browser to mediate calls to the client's file system allows us to address a variety of native file-systems since calls are made through the HTML5's FileSystem API.

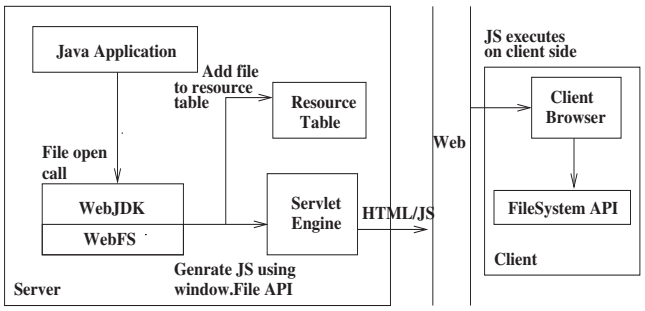

Figure 1. System Architecture

The WebFS system is based fully on CacioWeb and its session handling code is assimilated with that of CacioWeb. The WebFS sub-system includes customized implementations of the standard Java file access classes, predominantly the java.io class hierarchy. To understand the system better, Fig. 1 shows how different components of WebJDK come together. The design of WebFS is predominantly based on a partial implementation developed by the developers of CacioWeb [9].

The Java application running on the server issues a call for file system access, like a file open directive. The WebFS intercepts this call, the file to be opened is added to the list of resources. This allows us to keep track of open files and their associations. The WebFS generates appropriate JavaScript commands that are to be delivered to the client browser. The generated JavaScript uses the HTML5's File API to open the specified file. The web-page created by WebFS is delivered to the client though a Servlet engine. On receipt of the web-page by the client's browser, it executes the JavaScript code to open the specified file. Supported browsers will in turn call the concerned FileSystem API implementations on their systems with the specified parameters. This communication will be accompanied with a return value from the client in form of a POST reply that adheres to the return value of the function called in the java code as if it were executed on the client's device. Important functions of WebFS are detailed in the following sections.

A. Default Client Directory

Before any remote web application can access the local files, the client is needed to authorize each file access via. the HTML file input *<input type="file">* dialog or the HTML5 drag and drop method. This is vital in ensuring that the web application is not allowed full access of the client's file system. For use with WebFS, we require the user to demarcate a single directory as accessible. The

marked directory is imported by creating copies of all files and directories contained in the marked directory into the web application's *sandboxed* file system maintained by the browser which may or may not be persistent, however in our case we keep this file system as persistent [10]. The contents of the imported directory serve as a limiting field for the WebFS to access, all read directives are executed on this *sandboxed* file system; similarly all write operations to persistent storage in client's file system are done on the same file system. At the client's discretion, he may allow access to a single isolated directory or if need be, the client can expose higher hierarchies of directories for use by the web application.

For example, if a file open call is issued to the WebFS, the path to the file being specified in the call; HTML5's FileSystem API will resolve the file's location taking the selected directory as the root of the file system. Hence malformed paths to the file's location will not be resolved and developers should be aware of the nature of the directory the client would be marking; any expectations of already existing files that the program may open like logs etc. maintained by applications must be conveyed to the client and appropriate directory must be shared.

The inflexibility thrust upon the web application to be confined to the demarcated directory allows better security against unwarranted access of client's local files. A general workaround to this could be marking the whole *home* directory in case of Linux systems for WebFS; however the browser imposes constraints on the size of the *sandboxed* file system and it is very easy to spill over these limits if the whole system per se was to be shared. Thus it is at the client's discretion, that only functionally specific directories are exposed.

B. Resource Table

The resource table is designed to keep a track of currently open files that are being operated on or are being held in memory. The resource table helps with resolving file instances in the Java application with the corresponding files present in the *sandboxed* file system. Unlike direct system calls that are made on behalf of the JVM to operate with the file system, the WebFS re-directs them to the client. In such a system, it is the responsibility of WebFS to track currently open resources and provide references to them.

Hence for each file requested a tuple will be stored in the resource table to represent the open file. HTML5 interactions provide unique Uniform Resource Identifier (URI) for every open file or blob object [11]. This URI is used by the WebFS system in developing JavaScript code that references open files and directories in the Java application. This is particularly useful for handling closing of files which needs to be followed by flushing all operations on the particular resource. Also operations on files that are no longer present in the resource table should generate errors, in the same way when operations on file objects that are invalid results to a NullPointerException.

A typical tuple in the resource table can be defined with the following attributes:

<URI, Lock, Permissions>

Where:

- *URI* : Uniform Resource Identifier for Files and Blobs
- *Lock* : Lock parameter signifies whether an object is beign exclusively operated
- *Permissions* : Identifies the permissions associated with a particular file, if read-only, read-write etc

The *createObjectURL* method of the File API returns the *URI* associated with an open File object. Once files are opened using WebFS as the target file system, chances are that the same file may be referenced by another thread. In such cases, it is crucial that we identify file directives by cross-checking with the resource table to handle concurrent requests. The WebFS system is designed to be a thread safe system allowing multiple threads to safely call files without having to worry about system-wide visibility locks. Thread safety is achieved by *Lock* parameter which is an underlying semaphore that allows only one thread of execution to operate on a file. WebFS handles multiple requests by pipelining consequent requests, if a certain file is open, the lock for that file is entered and consequent requests will have to wait till the file is closed. Unlike the current Java I/O system, the WebFS provides a narrow and single line of execution. Thus it is crucial for the developer to close all files that have been opened through WebFS explicitly to avoid retaining the lock when the file is in use.

The *Permissions* parameter is developed to correlate permissions when a file is opened and the permissions afforded by the underlying *sandboxed* file system. In all operations on a file, the *permissions* parameter is cross-checked to validate the file access. The resource table is a reflection of the properties of the files stored on the client's *sandboxed* file system which resides at the server side.

C. File System Selection

WebJDK is envisaged to allow switching between WebFS and the native file system. This is important because systems often rely on drivers, sub-systems that require interaction with native storage. By leaving the choice of which file system to employ for file access, the developer can use a hybrid model of remote and native file storage. Also crucial software components such as drivers and loggers require native file access, mandating WebFS for all file accesses is not a good design choice.

978-1-4673-5758-6/13/$31.00 © 2013 IEEE

Typically the JRE specifies a native file system, all IO interactions are devised around this file system alone. To be able to specify a file system to be used for specific files, the *File* object constructor has been modified in WebJDK, the constructor accepts a *FileSytem* object. This object of type *FileSystem*, which is an instance of WebFS file system represents WebFS and its methods. The *File* object is then associated with the WebFS and all interactions with this object are conducted on the lines of the WebFS file system and not the native FileSystem.

The Java I/O hierarchy is not inherently constructed to support multiple file systems, this is evident by the fact that file system parameter in the *File* class of the *java.io* hierarchy is statically executed and defined as a static variable. To allow specification of file system dynamically, the file system attribute for *File* objects needs to be made non-static. Thus the file system attribute will not remain the same across all *File* objects and different *File* object instances will be associated with either the native file system or the WebFS. Further, classes that assist the reading and writing processes will need to identify the file system associated with a file object to direct the flow of execution to either the default methods of these classes or redirect the flow of execution to WebJDK.

V. Security Considerations

For a system that requires transfer of user files over the internet, confidentiality and authenticity must be ensured. The risks associated with such a system are often referred as file upload vulnerabilities. Under WebFS, files are read into slices of binary data or *blobs* and sent over to the server via. HTTP Post and such slices are assembled at the server side to complete the read request. Thus the data needs to be protected from any third party access while being sent from the client to the server. To prevent *man in the middle* like attacks, we advise the use of HTTPS for establishing connections between the client and the server.

With regards to the nature of the content being sent to the server, the current system is not directly vulnerable to file upload vulnerabilities as seen in technologies like PHP [12] because file data is read from the HTTP Post reply from the client and encoded appropriately and returned to the respective function call. Files are not written to the server system by default. Thus possible attacks like uploading executable files, uploading files with malicious paths for overwriting critical files, uploading HTML files that contain scripts to subject the victim to Cross-site Scripting (XSS) etc. are not inherently possible as files being read aren't written but held in memory.

However much depends on what the Java program running on WebJDK does with the read file contents. If a malicious file is read from the client and written to the server's local file system and executed, it is a direct realization of file upload vulnerabilities discussed earlier. Thus much of the responsibility of securing the system depends on the code being executed and developers must be careful of how files are being handled and the paths they might be written on. Critical files on the system can be easily replaced if the code running on WebJDK allows such. The current system lacks any method of preventing overwriting of critical files or policy guidelines to handle client data securely. This freedom in interacting with the files read over WebFS can prove self destructive if Java applications are written haphazardly.

The browser's *sandboxed* file system is unique to each web application that requests it and no two applications can access each other's *sandboxed* file systems. The browser maintains quota restrictions on the size of these file systems. Thus even though theoretically the WebFS system can be used to read arbitrarily large files from the client, which can cause overflow much like a file space denial of service attack, it is unlikely that the *sandboxed* file system's quota limits would not be violated while doing so. The hard restrictions on the size of the file system control such possibilities and we can tune the quota limits to suit our operational needs.

On the subject of writing files to the client through WebFS, the *sandboxed* file system prevents these files from being written anywhere outside its boundaries. The *sandbox* quarantines the files and prevents them from being executed. This prevents attacks on the client in the event of a malicious code being written to it with the intent of executing it on the client systems. The file system is also governed by the quota limits that specify maximum available space for use by an application. This protects the client systems from attacks orignating from malicious code executing on WebJDK. To draw conclusions from the above discussions, we see that the system is predominantly prone to vulnerabilities originating from malicious or unsafe code being executed on WebJDK. Thus it is imperative to establish guidelines for interacting safely with WebFS to overcome the system's inherent vulnerabilities.

VI. Fine-tuning WebFS

The effective speed of the network existing between the client and the server is a major factor governing the performance of WebFS for read and write calls. Uploading large files in one go can be slower at times. Also errors experienced during a file upload to the server may render all of the previous progress useless. This is a major concern in transmission of large files over the web. The WebFS performance is hugely affected by the time taken to upload or download a file. In a comparatively fast local area network, WebFS may experience no errors in transmission of files due to the relatively low traffic. Interestingly, the performance may hugely degrade in an slower or busier network. To allow us to leverage the best of the network bandwidth available to us, developers can fine-tune WebFS for the perfect size of the data chunk to be uploaded at a time.

978-1-4673-5758-6/13/$31.00 © 2013 IEEE

The HTML5 File API allows us to slice files into smaller chunks of binary data or *blobs*, thus instead of uploading files as is, we can slice them down in chunks of specified sizes and upload each of these chunks individually. This practice allows us to resume uploads in case of errors from where the upload was interrupted. Large files can be sliced down into manageable chunks for download as well as upload to server. Slicing of files into smaller chunks also speeds the transfer process as lesser errors occur since the chunk size is adapted to the bandwidth available.

The WebFS system provides an interface that allows the developer to set the appropriate chunk size for slicing files. All data transfers involving file data will be done in units of *blobs* of the specified chunk size. Thus the developer can tune the system to obtain the perfect chunk size needed to rightly leverage the available bandwidth in the deployment scenario. This fine-tuning can be either done by trial and error or approximation. As setups vary from the internet to institutional networks, the available bandwidth also changes. WebFS provides the developer to dynamically set the slice sizes and ensure that the data transfer takes place in the best interest of available resources.

VII. Browser Compatibility

WebFS relies on the FileSystem API being be supported by the client's browser. At the time of writing, Google Chrome, Google Chrome for Android and Blackberry browser fully support HTML5's FileSystem API [13]. The FileSystem API is a working implementation of an evolving specification by the World Wide Web Consortium (W3C) and may possibly undergo changes as the specification is further developed.

It would be wise to consider the fact that WebFS will only be able to interact with the newest of browsers. This compatibility requirement may exclude a major portion of users who run dated browsers or who have browsers which do not support the API. The API support is bound to grow with the widespread use of HTML5 to deliver high quality web applications to clients, especially in the mobile devices segment. The evolving nature of the API introduces a fair amount of unpredictability. However it is understood that the API will find support in future releases of leading web browsers.

VIII. Performance

To analyse the performance of WebFS, we must look at the system from three different facets. The first facet being the execution of the script on client side. The second facet being the transfer of the file contents or results back and forth between the client and the server and the third facet being the execution of WebFS code on the server. These three relatively uncoupled facets represent the processing overhead experienced during any WebFS call that requires interacting with the client.

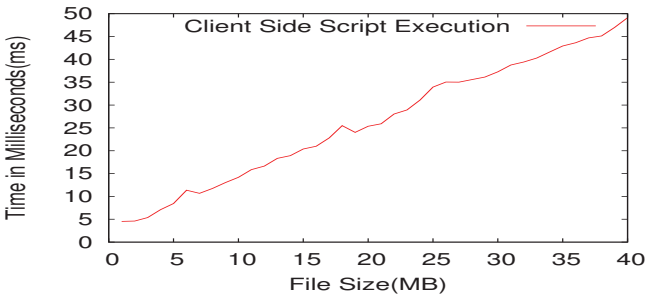

Figure 2. Client Side Script Execution Time

When a call for opening a file is received by WebFS, the appropriate file opening code is constructed by WebFS and the script is delivered to the client. The client receives this JavaScript and executes it in its browser. The script being executed is pure JavaScript and Fig. 2 gives us the performance graph for client side script execution for file read commands for various file sizes[1]. On monitoring the average time taken to reach each file we obtain the previously mentioned graph. We observe that the time taken grows linearly with file size.

After the reading of a file is complete the file data must be sent back from the client to the server running WebJDK. WebFS receives this data from the client through POST replies from the client. Hence a lot depends on how fast this transfer takes place. Data transfer time is the predominant source of running time overheads in WebFS. Performance of WebFS can be improved by specifying effective chunk sizes for slicing larger file to prevent it from overburdening the network with huge file transfers or underutilizing the available bandwidth. It must be noted that there is no inbuilt caching mechanism built in WebFS to cache a file read recently and serve the file if it is called again instead WebFS contacts the client for every interaction. Better performance can be achieved through caching of file requests and buffered reading and writing; this however will complicate the working of WebFS and also increase the memory requirements but is worth exploring. Once the transfered data reaches the server, WebFS encodes and returns this data to the called function; this has very little overhead to account for.

References

[1] Asar Jan Kashif Khan. Master's thesis, School of Computing, Blekinge Institute of Technology, SE-371 79 Karlskrona, Sweden, [Online] Available http://www.bth.se/fou/cuppsats.nsf/all/8731e09062623358c12578f000617240/$file/BTH2011Khan.pdf [Acessed: 17 January 2013], 2011.

[2] Amr Kamel Hani Samir and Eleni Stroulia. Swing2script: Migration of java-swing applications to ajax web applications. In *Reverse Engineering, 2007. WCRE 2007. 14th Working Conference*.

[3] OpenJDK. Caciocavallo project homepage. [Online] Available: http://openjdk.java.net/projects/caciocavallo/. [Accessed: 3 January 2013].

[1]Results obtained on device using Intel i3 processor and 3GB memory running Ubuntu 12.04

[4] LadyBug Studio Mario Torre. Cacioweb and webjdk: deploy your java applications everywhere. In *Java Embedded Technology Conference*, Munich, Germany, February 2012. JET-CON 2012. [Online] Available: http://www.jet-con.eu/?q=node/32 [Accessed: 2 January 2013].

[5] Roman Kennke. Caciocavallo 1.1 released. [Online] Available: http://rkennke.wordpress.com/2012/05/02/caciocavallo-1-1-released/ [Accessed: 3 January 2013], May 2012.

[6] W3C Working Draft. *File API: Directories and System*. W3C, April 2012. [Online] Available: http://www.w3.org/TR/file-system-api/ [Accessed: 2 January 2012].

[7] Wikipedia. Virtual network computing. [Online] Available http://en.wikipedia.org/wiki/Virtual_Network_Computing [Acessed: 17 January 2013].

[8] Roman Kennke. Cacioweb - the java deployment solution of the future. [Online] Available: http://rkennke.wordpress.com/2011/11/01/cacioweb-the-java-deployment-solution-of-the-future/ [Accessed: 3 January 2013], November 2011.

[9] Roman Kennke LadyBug Studio. Partial webfs implementation. [Online] Available: ladybug-studio.com/~roman/webfs.tar.gz [Accessed: 16 January 2013].

[10] Eric Bidelman. *Using the HTML5 Filesystem API*, chapter 4. O'Reilly-Google Press, 2011.

[11] W3C. A uri for file and blob reference. [Online] Available: http://www.w3.org/TR/FileAPI/#url [Accessed: 3 January 2013], October 2012.

[12] CERT. Multiple file upload vulnerabilities in php. [Online] Available: http://www.cert.org/advisories/CA-2002-05.html, [Accessed: 17 January 2013], February 2002.

[13] caniuse.com. Htm5 filesystem api support in leading web browsers. [Online] Available: http://caniuse.com/filesystem, [Accessed: 17 January 2013].

978-1-4673-5758-6/13/$31.00 © 2013 IEEE