Anywhere Apps:

JavaScript and Browser Ubiquity

Avery L. Erwin-McGuire

The University of Massachusetts Amherst

Abstract

Anywhere Apps is a research project currently being developed by Tim Richards and Emery Berger at the University of Massachusetts. The focuses of the project include the division between the client and the server, the browser serving as a universal operating system, and JavaScript functioning as an assembly language. Presently, research participants are developing a client-side cache which utilizes a server as main memory.

Discussion

Initially the research was largely concerned with compiled JavaScript. It was thought that JavaScript could be thought of as an assembly language for the web (Zakai, 2011). As the development of JavaScript was rushed, it has certain odd characteristics which can be difficult to comprehend and work with. One example of this is scoping, unlike most modern languages JavaScript only has a global and functional scope. This can lead to tremendous namespace pollution and unexpected errors. The obscure design of this language is similar in this way to assembly, where commands relate directly to hardware components and mystify those who are not well versed in the architecture. Planning out and hand writing a program in assembly is not a common activity; such tasks are left for compilers.

There are several available JavaScript compilers, including GWT, Pyjamas, Emscripten, and Closure. Much of the research focused on code produced through GWT. Through writing out simple programs and considering the output, it was deduced that GWT simply matched up the functionality available in JavaScript to existing Java libraries. Certain libraries which exceeded the capabilities of JavaScript were simply disallowed within the project. Upon compile, the code was optimized through the v8 engine and produced JavaScript code nearly unreadable to humans. This result was reminiscent of programs written in assembly, and gave further evidence to the research idea.

With high level languages, programmers need not consider the intricacies of their machine and its architecture. Tasks such as fetching data, clearing registers, and minding the stack are left to the domain of the compiler and its assembly. The complexity of the system is hidden by abstraction. The web is increasingly becoming a new system to itself, with the browser behaving as an operating system. However, the current structure of the web requires that programmers remain aware about the physical structure of the system for which they write. Developers must be mindful of and write code which reflects the divide between the client and the server – even though that divide is ultimately physical in

nature. Utilizing JavaScript as an assembly language for the web, one could abstract away the architecture and leave programmers able to write code in whichever language they desire, only concerning themselves with proper programming paradigms. Moreover, as the web grows it becomes increasingly difficult to keep up with the fast paced rate of development(Silva, 2012), and compilers could help with such lag time.

Researchers began looking into producing a client side cache, where the server functioned as main memory. This utilized HTML5 local storage, which saves an array of string key/value pairs within the browser. Within a single machine, there are physical divisions between cache and main memory. This can be seen as a parallel to the division between the client and the server. Therefore by implementing a client side cache, it can be shown that the underlying structure of the web can be abstracted away.

The program had several components: the client, the server, and the communication between the two. Researchers aimed to formalize the messages sent across the network in order to make them resemble calls for data within a single machine. These calls were known as “readLine” and “writeLine” and were GET and POST requests respectively. The commands allowed the communication and manipulation of “file” objects, which were a very simplified version of their namesake. Within the scope of the project, a file consisted of a single character as a file name such as “A” or “C” and four lines of text. The server’s file system contained the files “A”, “B”, “C”, and “D”.

Upon loading the page the client had access to a simple interface to load a file to the screen, edit it, and save the changes. When a file was requested, the page’s JavaScript code would attempt to load the lines “C0” “C1” “C2” “C3” where C is the name of the file. This would occur by querying local storage, which is a key/value array. If the value did not exist locally, the client would send a “readLine” GET request to the server which carried all the relevant data within a URL parameter.

In order to further simulate a cache, there was an artificial limit of 10 values imposed on the local storage. This meant that upon reaching 10 values, the client must begin to release data in order to store new values. Local storage sorts its keys alphanumerically rather than by the time of assignment. This meant that researchers had to simultaneously push the key values to a queue so as to know which values were the oldest, and so were to be release from local storage. As the readLine request returned the relevant line to the client, it was saved into local storage. The saving process included releasing the oldest stored data once the size limit was reached. When data was stored locally the server was not contacted. This implemented the efficiency of a cache (Franklin, Carey & Livny, 1997).

When the user selected for some modified data to be saved, the client would check to see which values entered differed from those in storage. The modified lines were communicated to the server with a “writeLine” POST request, where the data was carried in URL parameters.

When the server received any communication from the client, it would identify the request as either being a JavaScript or HTML request, or a command. Commands took the form “GET readLine/” or “POST writeLine/” . The parameters of these requests held the file name, the line number, and in the case of writeLine the data to be stored. The file system was then accessed and the relevant actions were taken. A confirmation or the data requested would then be sent back to the client.

There were several problems with this model, and in the future these problems will be addressed and improvements made. One factor researchers will consider is the form of the request. The code currently utilizes synchronous AJAX requests, which could be changed to asynchronous in order to save time. Additionally, the method to save and release data is inefficient. This is because the client does not take into account which file it is currently trying to load, and will begin to release lines of the file locally if it is the oldest data. This leads to necessary data being released and re-requested. In the future, researchers will insert new code to prevent this case.

Researchers plan to extend this project once the problems listed above are addressed. There are several possible features which will be included in the next version of the cache. These may include more diversity and complexity in the file format, or perhaps only serving bytes rather than strings. The user may be able to create files locally and have these files communicated to the server in a single “writeFile” request. Another avenue under consideration is handling the case where multiple users may be accessing the server’s file system simultaneously. This would require the programmers to implement a version history documenting the editing of files, so as to ensure the client’s cache is up to date.

References

Franklin, M., Carey, M., & Livny, M. (1997). Transactional client-server cache consistency: Alternatives and performance. *ACM Transactions on Database Systems*, *Vol. 22*(No.3), Pages 315–363.

Silva, C. (2012). Reverse engineering of gwt applications. *EICS'12*,

Zakai, A. (2011). Emscripten: An llvm-to-javascript compiler. *SPLASH'11 Companion, Portland, Oregon, USA.*,