Computer Science 4327

Dr. Smith

April 25, 2018

Nathan Robinson

Assignment #6: Summary: The Gap Between O.S. and Browser

This is a summary of a paper called “BROWSIX: Bridging the Gap Between Unix and the Browser,” written by Bobby Powers, John Vilk, and Emery D. Berger, of the University of Massachusetts Amherst. They created Browsix. It is a framework that uses a shared kernel and system call conventions to unite Unix abstraction with browsers.

The motivation was that browsers good at building interfaces, but are not platforms to build sophisticated applications. Running such applications on the traditional browser requires heavy modification because browsers do not support shared file system, libraries, processes, or even other synchronized programs. This is why there is a division between front end user interface development in browsers and backend server development on an operating system like Unix.

Browsix consists of three major parts. First is a Javascript-only operating system kernel. This kernel offers the core services of a full OS that applications expect. Second is a range of system call abstractions. Third are the processes that emulate Unix via Web Workers.

The core of BROWSIX’s OS support is a kernel that controls access to shared Unix services. Processes run separately and in parallel inside Web Workers, and access Browsix kernel services through a system call. The kernel is an intermediary between processes and loosely coupled Unix subsystems. Unix services that run on the kernel include the shared file system, pipes, sockets, and task structures. Browsix exposes standard pipe API, making it simple for developers to compose process into pipelines. Concurrent processes are also support. Process-related system calls like fork, spawn, exec, wait4 act as process primitives on top of Web Workers, letting applications run in parallel and spawn subprocesses. Browsix supports a subset of POSIX signals API. Kill and signal handlers can also be employed to let processes communicate with each other asynchronously. TCP socket servers and clients are supported to make it possible to run server applications like databases and HTTP servers together with their clients in the browser. Browsix even offers a type of shared file system. Processes can share their state through a shared filesystem. Lastly, Browsix is language agnostic. It is integrated with the runtime libraries libraries of Emscripten and GopherJS. There are also extended JS runtimes for C, C++, Go, and Node,js that let unmodified programs in these languages (and compiled in JS) to run directly in the browser. Using Browsix requires no plugins and applications can run in a wide range of unmodified web browsers like Chrome, Edge, Safari, and Firefox.

System calls on Browsix can be asynchronous and synchronous. Asynchronous system calls have a high performance penalty on C and C++ programs, but work in all modern browsers.Unix orocesses on Browsix rely on Web Workers. A Web Worker has access to only a subset of browser interfaces, runs in a separate execution context, and can only communicate with the main browser context via asynchronous message passing.

Browsix also offers runtime support. Applications invoke Browsix system calls indirectly through their runtime systems. GopherJS, Emscripten, and Node.js have added support with the APIs and web applications so they can execute programs in Browsix. This includes browser environment extensions for web applications to have access to Browsix features through several global APIs. Browsix provides a small syscall layer that runs in a Web Worker. This concrete API for asynchronous system calls over the browser’s message passing primitives enables language runtimes to use this layer to communicate with the kernel. Programming language runtimes targeted for the browser must handle the mismatch between synchronous APIs on Unix-like systems and the asynchronous browser environment. Because Browsix supports both synchronous and asynchronous system calls, language runtimes can choose the most appropriate system call convention.

The authors evaluated their system based on two questions. First, does bringing Unix abstractions into the browser enable compelling use cases? In the first case study Browsix was used to build a web application for creating memes. The meme generator can run its unmodified server in Browsix. Thus it transparently switches between generating memes in-browser or server-side depending on network and device characteristics. Second, we build a Unix terminal that lets application developers use dash, a widely-used POSIX shell, to interact with Browsix in a familiar manner. Based on the two case studies and the Latex editor that were built, clearly Browsix-like systems simplify porting applications to browser environment by using Unix abstractions.

Second, is the performance impact of running programs under Browsix acceptable? Browsix’s performance is limited by the performance of underlying browser primitives, it provides acceptable performance for a range of applications. Browsix’s performance in the case studies and with microbenchmarks show that its overhead is low enough for real-world usage.

In conclusion, creating Browsix emphasized areas tjat need improvement in the specification and design of Web Workers and other browser APIs. Optimizations and natural extensions that would extend Browsix’s reach include allowing for worker priority control, reducing postMessage() backpressure, improving message passing performance, support for memory mapping, and C-like stack management*.* The development of Browsix also contributes OS abstractions and services to the Browser on top of web APIs, runtime integration for languages, and useful case studies. Browsix is a framework that brings the essence of Unix to the browser. Browsix makes building complex web applications from components very simple. Components can even be written in a variety of languages and no code modification is necessary. Browsix significantly reduces the effort required to build highly sophisticated web applications.

BROWSIX: Bridging the Gap Between Unix and the Browser

Bobby Powers, John Vilk, Emery D. Berger

University of Massachusetts Amherst

{bpowers, jvilk, emery}@cs.umass.edu

# Abstract

Applications written to run on conventional operating systems typically depend on OS abstractions like processes, pipes, signals, sockets, and a shared file system. Porting these applications to the web currently requires extensive rewriting or hosting significant portions of code server-side because browsers present a nontraditional runtime environment that lacks OS functionality.

This paper presents BROWSIX, a framework that bridges the considerable gap between conventional operating systems and the browser, enabling unmodified programs expecting a Unix-like environment to run directly in the browser. BROWSIX comprises two core parts: (1) a JavaScript-only system that makes core Unix features (including pipes, concurrent processes, signals, sockets, and a shared file system) available to web applications; and (2) extended JavaScript runtimes for C, C++, Go, and Node.js that support running programs written in these languages as processes in the browser. BROWSIX supports running a POSIX shell, making it straightforward to connect applications together via pipes.

|  |
| --- |
| We illustrate BROWSIX’s capabilities via case studies that demonstrate how it eases porting legacy applications to the browser and enables new functionality. We demonstrate a BROWSIX-enabled LATEX editor that operates by executing unmodified versions of pdfLaTeX and BibTeX. This browseronly LATEX editor can render documents in seconds, making it fast enough to be practical. We further demonstrate how BROWSIX lets us port a client-server application to run entirely in the browser for disconnected operation. Creating these applications required less than 50 lines of glue code and no code modifications, demonstrating how easily BROWSIX can be used to build sophisticated web applications from existing parts without modification.  Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.  *ASPLOS ’17* April 8–12, 2017, Xi’an, China.  © 2017 Copyright held by the owner/author(s). Publication rights licensed to ACM.  ISBN 978-1-4503-4465-4/17/04...$15.00  DOI: http://dx.doi.org/10.1145/3037697.3037727 |

# Introduction

Web browsers make it straightforward to build user interfaces, but they can be difficult to use as a platform to build sophisticated applications. Code must generally be written from scratch or heavily modified; compiling existing code or libraries to JavaScript is not sufficient because these applications depend on standard OS abstractions like processes and a shared file system, which browsers do not support. Many web applications are thus divided between a front-end UI that runs in the browser and a backend server. The backend runs on a traditional operating system, where the application can take advantage of familiar OS abstractions and run a wide variety of off-the-shelf libraries and programs.

As a representative example, websites like ShareLaTeX[[1]](#footnote-1)and Overleaf[[2]](#footnote-2) let users write and edit LATEX documents in the browser without the need to install a TEX distribution locally. This workflow lowers the barrier for students and first-time LATEX authors and enables real-time collaboration, eliminating some of the complexity of creating multi-author documents. These applications achieve this functionality by providing a browser-based frontend for editing; user input is sent to the server for persistence and collaboration purposes. When the user requests a generated PDF, the website runs pdflatex and bibtex server-side on the user’s behalf, with the resulting PDF sent to the browser when complete.

1. <https://www.sharelatex.com/> [↑](#footnote-ref-1)
2. <https://www.overleaf.com/> [↑](#footnote-ref-2)