Generic Workers – Towards Unified Distributed and Parallel JavaScript Programming Model

Adam Welc Richard L. Hudson Tatiana Shpeisman Ali-Reza Adl-Tabatabai

Intel Labs

{adam.welc,rick.hudson,tatiana.shpeisman,ali-reza.adl-tabatabai}@intel.com

Abstract

In this paper we introduce *generic workers*, a programming model for JavaScript unifying parallel and distributed computing paradigms, that allows the same application to run well on a variety of clients while utilizing the available resources in the best possible way. We describe the design and implementation of an infrastructure supporting our programming model and evaluate performance of selected applications running on devices with differing computational capabilities.

1. Introduction

Recently it has become quite clear that the era of performance improvements due to steadily increasing CPU clock speeds is over and, consequently, programmers must turn to distribution and/or parallelism to further improve performance of their applications. Surprisingly, despite growing acceptance of the web browser as the dominant application delivery system and JavaScript as the language to implement complicated and often resource-demanding web applications, support for distributed and parallel programming in JavaScript remains limited.

Web workers offer one way to exploit parallelism in JavaScript. Its draft specification [6] is part of the HTML5 standards [2] effort and while it was designed as a mechanism to improve latency, its implementations in some of the existing web browsers (e.g. Google Chrome) can be used to implement scalable parallel applications. By design, however, web workers lack any support for computation distribution. Consequently, todays programmers must either rely on web workers for local parallelism or write applications explicitly tapping to the remote resources available in the cloud. As a result of the lack of uniformity between these two programming styles, a parallel application that runs well on a high-performance multi-core machine might not run well on a less powerful hand held machine. At the same time, by relying only on remote resources todays web programmers may forgo a potential benefit of parallelism available on a client platform.

In this paper we propose *generic workers*, a programming model generalizing web workers to allow workers to execute not only locally in a web browser but also remotely on a *compute server* (described in more detail in Section 2.1). Generic workers combine benefits of local parallel execution with those of the remote distributed execution. As the programming model is the same for both local and remote workers, the same code can be used for a generic worker regardless of where it ends up being executed. This way, the best execution mode for a given application can be chosen based on the current resource availability with virtually no changes to its source code. We also describe an implementation of generic workers support and present its performance evaluation.

2. Unified Programming Model

In JavaScript, communication between different parallel or distributed entities is realized via message passing. However, message passing is handled differently, depending on whether communication happens between the web workers (parallel setting) or between JavaScript code and a remote server (distributed setting).

Conceptually, web workers are threads of execution that communicate exclusively by exchanging messages. A message can be sent to a web worker by invoking the `postMessage` method on this worker's instance. On the receiving side, a worker defines an `onmessage` handler that gets invoked by the browser to notify this worker of the message received event. The receiving worker, as part of executing the message event handler, can send response messages to the original sender also via the `postMessage` function.

JavaScript's mechanism for communicating with remote servers (typically HTTP servers) is quite different, as server

```
**** CREATION AND INTERACTION ****

GenericWorker(worker_source, handler, address, port);

GenericWorker.post(data);

GenericWorker.destroy();

**** PROGRAMMING ****

onget(data) { ... }

post(data);
```

Figure 1. Generic workers API

communication is supported via HTTP protocol [5]. An `XMLHttpRequest` object [1] encapsulating an HTTP request (e.g. GET or POST) must be created before a request message can be sent to the HTTP server. The HTTP server must be configured appropriately to understand incoming request messages and send HTTP response messages that are then passed to the original sender via event notification.

Our unified programming model, generic workers, is consistent with the spirit of JavaScript's in that it supports message delivery via asynchronous notifications, but removes the source of the dichotomy described above to facilitate code re-use and in doing so allows application to adapt to varying resource availability.

2.1 Generic workers

The main idea behind the generic workers is to allow the same application programming interface (API) to be used for programming both parallel and distributed applications. During its creation, a generic worker is specified to be either local or remote. Except for this, the same generic worker code can be executed without any changes on any platform - locally inside a browser or remotely on a server, which we call a *compute server*. A compute server is essentially an execution engine for web workers running on a remote

```
function wh(data) {
  PRINT(data.str); this.destroy();
}

function local_main() {
  var w = GenericWorker("ew.js", wh, null, null);
  w.post({"str":"STRING TO BE ECHOED"});
}

function remote_main() {
  var w = GenericWorker("ew.js", wh, "addr.net", P);
  w.post({"str":"STRING TO BE ECHOED"});
}

function main() {
  local_main(); remote_main();
}
```

Figure 2. Echo application code

machine, that supports execution of both standard JavaScript and the API specific to generic workers.

The generic workers API, presented in Figure 1, is divided into two components, one defining how to create and interact with the generic workers and the other defining how to program a generic worker itself.

The first API component defines a constructor for creating a generic worker instance. An address and a listening port of a compute server must be specified if a generic worker is to be instantiated remotely. If both of these arguments are `null` then a local worker is created instead. Though currently not implemented, in the future we envision a resource discovery mechanism that will instantiate workers automatically at appropriate locations depending on resource availability. Even without this functionality, however, programmers using our API may offer the end-users the benefit of semi-automatic adjustment to the current execution environment. For example, in an image processing application, a click of a button may allow the end-user to decide if the image should be processed locally or remotely. Other generic worker constructor parameters include the source used to instantiate the worker and a handler function to be

```
importScripts("generic_workers_api.js");

function onget(data) {
  post(data);
}
```

Figure 3. Echo worker code (ew.js)

executed when a message from this worker arrives and needs to be processed. Interaction with a generic worker's instance is possible via the `post` function used to send a message to a generic worker and via the `destroy` function used to destroy a generic worker's instance (local or remote).

The main element of the second API component is a handler function `onget` that must be defined for every generic worker. Conceptually, a newly constructed generic worker is idle until a message arrives and the runtime invokes the handler function. A generic worker's handler function typically does some computation that might use the `post` function to send a message back to the sender. Upon completion of the handler function the generic work returns to the idle loop waiting for another message.

In Figure 2 and in Figure 3 we present a simple JavaScript "echo" program that uses the generic workers API (`PRINT` function is used by the browser to print to the screen, `P` represents compute server's port number, and we assume that file "ew.js" is available both to the browser and the compute server). After both the local worker and the remote worker are instantiated in the `main` function, using *the same* source code (Figure 3), they wait for arrival of a message just to pass it back to the sender. Please note that data marshalling/unmarshalling happens automatically, regardless of whether interacting with a local worker or a remote worker.

3. Implementation

Our implementation consists of two major components - the *local component* supported within the browser and the *remote component* supported by the compute server. The local component is responsible for creating and interacting with both local (parallel) generic workers and remote (distributed) generic workers, but executes the computation of only local workers. It is the remote component, a compute server, that is responsible for executing computations of remote generic workers.

The implementation of the local component utilizes custom-built NPAPI (Netscape Plugin API [7]) plugin to provide some necessary native support that was missing from the browsers at the time we were building our infrastructure. The implementation of the remote component is coded entirely in JavaScript and executed on top of v8cgi [10], a publicly available wrapper around Google's V8 JavaScript execution engine [9] providing a standalone JavaScript execution environment. Due to space considerations, we provide only a sketch of the implementation.

We implemented communication between the local component and the remote component using "raw" TCP protocol [8], with data being marshalled/unmarshalled under-the-hood using JSON [4]. We used the NPAPI plugin to access TCP sockets as the implementation of WebSocket-s [3] was not available at the time we were building our infrastructure [1]. An infrastructure supporting execution of local generic workers is mapped to web workers, but with an interesting caveat related to the fact that workers can create other sub-workers. If a local worker attempts to create a remote sub-worker, things get more complicated as the local worker [2] does not have direct access to any data defined in the main program or in the main browser window. As a result, a local worker cannot access the plugin object, which means that it also cannot directly create a remote sub-worker or interact with it. We overcome this obstacle by introducing a level of indirection. Conceptually, local workers form a tree hierarchy with the main browser thread at the root and local workers as the tree nodes. A local worker wishing to create a remote sub-worker or to interact with it passes special control messages up the tree until they reach the root (which in turn can directly access the plugin).

The main part of the remote component (i.e. compute server) is a loop whose sole purpose is to receive incoming messages. The first message received by the compute server on the server TCP socket [3] carries a request from a client (either a browser of another compute server) to create a remote worker on this server, and includes the name of the source file to be used for worker instantiation. Once this message is received, the content of the appropriate JavaScript source file is retrieved and evaluated (using JavaScript `eval` function) in the global scope, thus becoming accessible to the compute server's code (e.g. for the compute server to be able to execute a message handler of the generic worker being created).

4. Evaluation

We evaluate the performance of our system on an in-house parallelized RayTrace benchmark from Google's V8 benchmark suite, rendering images of three different sizes (in pixels): 100x100, 200x200 and 500x500. Our experimental setup consists of a local platform to execute local

[1] It is also possible to build an JavaScript-only implementation that wraps messages in the `XMLHttpRequest` objects and does not access sockets directly. As we discovered, however, such implementation is very fragile due to security-related workarounds required to support communication with multiple compute servers.

[2] More precisely, it is a local generic worker mapped to a web worker.

[3] Direct access to TCP sockets is provided by v8cgi's API.

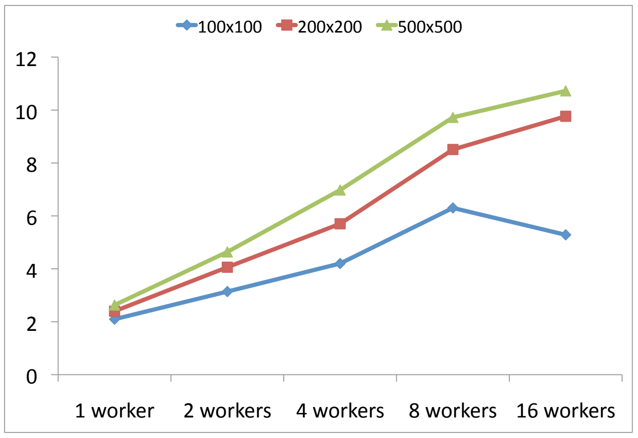

Figure 4. Remote speedup

part of the computation (in the Google Chrome browser v.4.1.249.1042 [4]) and a remote platform to execute remote part of the computation (on a compute server or multiple compute servers running on top of v8cgi using V8 engine v.1.3.16). We evaluate two different local platforms, a single-core Atom-based netbook with 1GB of RAM running Windows XP at 1.66Ghz and a 24-core Dunnington-based server machine with 16GB of RAM running Windows Server 2003 at 2.66Ghz. A twin to the latter machine, running RedHat Enterprise Linux 4 on top of 2.6.9-42.ELsmp kernel, is used for remote execution of compute servers in all cases.

As we layered local implementation of generic workers on top of web workers, and Chrome does provide a scalable implementation of web workers, our workload implemented using parallel generic workers scaled well when executed on a Dunnington-based machine. Without any extensive tuning, we have achieved up to over 7x speedup when using 16 workers, which is the maximum number of workers supported by Chrome. A more interesting question was whether the application can benefit from computation offloading, despite relatively high cost of communication (we used VPN-ed wireless connection with average round-trip time of 30ms). As we can see in Figure 4, presenting speedup of our workload offloaded from the Atom-based netbook to the Dunnington-based server compared with the sequential execution time of the same workload on the netbook, offloading application running on a netbook results in performance improvement even in the case when the entire computation is performed by just one remote worker. We can also observe that the remote configuration scales, as performance improves even further as additional remote workers are added to the mix.

[4] It was the fastest web browser (in terms of JavaScript execution) at the time we performed our experiments.

References

[1] World Wide Web Consortium. XMLHttpRequest working draft, 2009. http://www.w3.org/TR/XMLHttpRequest/.

[2] World Wide Web Consortium. HTML5 working draft, 2010. http://dev.w3.org/html5/spec/Overview.html.

[3] World Wide Web Consortium. The websocket api, 2010. http://dev.w3.org/html5/websockets/.

[4] Douglas Crockford. The application/json media type for javascript object notation (JSON), 2006. http://tools.ietf.org/html/rfc4627.

[5] Network Working Group. Hypertext transfer protocol – HTTP/1.1, 1999. http://tools.ietf.org/html/rfc2616.

[6] Web Hypertext Application Technology Working Group. Web workers draft recommendation, 2010. http://www.whatwg.org/specs/web-workers/current-work/.

[7] Mozilla. Gecko plugin api reference, 2009. https://developer.mozilla.org/en/Gecko_Plugin_API_Reference.

[8] Darpa Internet Program. Transmission control protocol, 1981. http://tools.ietf.org/html/rfc793.

[9] V8 team. V8 JavaScript engine, 2010. http://code.google.com/p/v8/.

[10] Ondrej Zara. v8cgi, 2010. http://code.google.com/p/v8cgi/.