WebMesh: A Browser-Based Computational Framework for Serverless Applications

Joseph Romano (joseph_romano@brown.edu)
Advisor: Rodrigo Fonseca (rfonseca@cs.brown.edu)

May 1, 2019

Abstract

WebMesh is a browser-based computational framework for serverless applications. Unlike existing serverless frameworks which run in managed datacenters, WebMesh nodes run in browser tabs, using WebAssembly to allow C++ code to run portably in web browsers and WebRTC to allow nodes direct peer-to-peer communication. We evaluate the scalability and feasibility of such a system by implementing the MapReduce distributed computing model on the framework. We find that, in tests of a word count job running on MapReduce, WebMesh is able to handle 200,000+ concurrent jobs, execute jobs 3.6 times faster than running locally, and support a variety of tasks. We suggest it is possible to create value from otherwise unused computing power, allowing more cost-efficient distributed computing than existing serverless providers.

1 Introduction

With the increasing ubiquity of personal computing devices which continue to grow in power, a large amount of processing power remains primarily idle, within smartphones in people's pockets or within computers on their desks. While there are many frameworks for distributed computing, commercial and non-commercial, none have lowered the barriers to entry to a simple task like opening a web page. With over two billion installations of Google Chrome worldwide, there is a massive opportunity to create value from preexisting and unused computing power, filling gaps in time between peak usage [23].

To fill this gap, we present WebMesh, a browser-based computational framework for serverless applications. WebMesh is able to use idle computing time by executing distributed applications on any environment that can run an up-to-standards web browser such as Google Chrome. We initially implemented WebMesh with MapReduce and tested using the canonical word count use case for MapReduce. This paper evaluates the scalability, adaptability, resilience despite unreliable workers, and versatility of such a system.

WebMesh advances existing research into browser-based, volunteer, and serverless computing by using WebAssembly to allow C++ code to run in a portable environment, and by using WebRTC for peer-to-peer transmission of data, a first step towards decentralization.

The remainder of this paper is organized as follows. Section 2 provides pertinent background information on serverless computing, WebAssembly, and WebRTC. Section 3 details the design decisions, execution model, and programming model. Section 4 evaluates the system empirically and subjectively, provides a brief financial analysis, describes shortcomings, and lays out ideas for future work. Related work is reviewed in Section 5. We conclude in Section 6.

2 Background

2.1 Serverless computing

Cloud computing has grown in popularity over the past few decades and providers have slowly shifted to manage additional services on behalf of customers. Virtual machines and containers were the

first wave, abstracting the need to manage machines, operating systems, and execution environments to some extent. Recently, serverless computing has become increasingly popular as a way of abstracting nearly every aspect of cloud computing, besides the actual code that is to run. Serverless computing providers manage the environment, provisioning, and more, promising customers that the code they submit will scale based on demand. This is advertised as not only easier for customers, but more cost efficient due to fine-grained billing.

One of the first mainstream providers of serverless computing, Amazon Web Services, markets their product as AWS Lambda, advertising "continuous scaling" and offers an event-driven service [1]. Lambda will run based on particular triggers, such as a request to an HTTP endpoint or other triggers from applications within AWS. Each Lambda worker is limited to only 15 minutes of execution, up to 3,008 MB of memory, and has no default support for inter-process communication (i.e. from Lambda to Lambda). Lambda supports a variety of languages such as Python, Java, Go, C#, and Ruby. In total, AWS Lambda provides an event-based serverless offering to fill a specific niche, and not necessarily for general computing. A framework developed to run on AWS Lambda offering more general distributed computing power, Pywren, is discussed in Section 5.

Google also offers a serverless computing architecture, not only providing event-based "Cloud Functions" similar to AWS Lambda, but also is alpha-testing running containers and Kubernetes on their serverless framework to provide the scaling and pricing benefits of serverless in more traditional settings [13]. This opens up more opportunity for general computing than AWS's offering.

2.2 WebAssembly

WebAssembly is an instruction set designed for languages such as C++ to run portably in web browsers or other environments [20]. Already implemented in major web browsers, these browsers' virtual machines are able to execute WebAssembly just like JavaScript is currently executed. The virtual machine takes in a WebAssembly binary format file, provides it any requested imports from JavaScript functions, and the binary file specifies a set of exported functions that the JavaScript code is able to call. The instantiated WebAssembly binary manages its own memory but is sandboxed in the same way that browsers currently sandbox JavaScript code.

This sandboxing provides the same security guarantees that existing JavaScript sandboxes provide, ensuring that malicious WebAssembly modules could only get so far as a malicious JavaScript file could. While it is out of the scope of this project to explore the security ramifications of WebAssembly, there is a baseline assurance that WebAssembly provides similar security promises as JavaScript.

WebAssembly also provides performance that can be close to native, with an average of an 89% slowdown and peaking at a 3.14x slowdown in Google Chrome on the SPEC CPU benchmarks [21]. Given that there could be dozens or hundreds of workers dividing time on a job, even the maximum slowdown of 3.14x would provide fast performance in a highly distributed setting in a system with low overhead. This performance will continue to increase as WebAssembly develops and matures.

While still novel, WebAssembly is able to integrate with existing compilers and toolchains to allow existing optimizations and libraries to be used on the web. Support exists for the newest versions of C++ using Emscripten, a toolchain branched off of LLVM [8]. Emscripten supports WebAssembly binaries by providing them the needed "glue code" with native JavaScript implementations of syscall, file system, I/O, and library functions. This is particularly important as WebAssembly runs entirely sandboxed, and must provide all of its own library code – even syscalls must be re-implemented in JavaScript to be imported into the WebAssembly module. Also, Emscripten manages the memory allocation given to the WebAssembly modules, growing the memory allocated in JavaScript whenever syscalls are made which increase memory pressure.

Emscripten also provides a tool, Embind, for easily accessing C++ functions and objects from JavaScript. This makes interoperability easier when providing input data to C++ functions from JavaScript. Furthermore, Embind provides access to class constructors and creates analogous JavaScript objects matching C++ objects (see Figure 1). This lets C++ functions take in and return C++ objects, so

```
emscripten::value_array<pair<string, int
    >>("PairStrInt")
    .element(&pair<string, int>::first)
    .element(&pair<string, int>::second);
```

Figure 1: Using Embind to match a C++ `std::pair` type to a JavaScript array

long as proper bindings are made to JavaScript types using Embind.

Another web technology important for WebAssembly is Web Workers [14]. Web Workers allow background tasks to run independent of the main event loop, ostensibly creating "multithreading" in JavaScript. Without Web Workers, long-running WebAssembly tasks would potentially block the main thread, thus freezing the entire webpage. By using a Web Worker, the WebAssembly code can be executed in the background, leaving the main page free to request data, take input from the user, and more.

2.3 WebRTC

WebRTC is a browser technology allowing for direct inter-browser networking and communication. While traditionally, web messages would require a server to process relay information as separate requests between two clients, WebRTC allows these clients to send packets directly to each other [15]. The primary use cases for WebRTC are real-time video and gaming, but the APIs also allow for general data transfer between any two clients.

WebRTC is not necessarily decentralized – it relies on a signalling mechanism on connection startup, whether by manually exchanging messages or by using a server. This is done to negotiate a connection between the two endpoints, and after this point all data can be sent directly between the two peers.

There are two supporting technologies common in prior VOIP implementations that are also used in WebRTC: STUN and TURN. STUN, or Session Traversal Utilities for NAT, helps WebRTC nodes determine their own IP address and port from within a NAT [27]. TURN, or Traversal Using Relays around NAT, is an extension of STUN and serves as a relay server in cases that the NATs of each host do not allow direct communication between each other [26]. When using WebRTC, it is necessary to have a server offering STUN and TURN services to ensure complete reachability. This creates significant cost overhead for a service using WebRTC, as TURN servers could potentially be forced to relay all peer traffic in the case of highly restrictive NATs or firewalls.

3 Design

The system is broken up into four major parts: the coordinator server, submitters, intermediary nodes, and workers. Submitters input C++ code and select input files, the coordinator manages tasks and relays some data and information between submitters and workers, intermediary nodes optionally relay results between dependent workers, and workers receive jobs from the coordinator and compute results. A separate WebRTC STUN and TURN server is used to assist in WebRTC establishment and reachability. An overview of the architecture and job lifecycle can be seen in Figure 2.

3.1 Coordinator

WebMesh relies on a central coordinator for compiling code to WebAssembly, job scheduling, negotiating WebRTC connections via the signalling mechanism, and optionally holding intermediary data in between compute phases. The coordinator is implemented in Node and uses Web Sockets to communicate with workers, intermediary nodes, and submitters.

- **Compilation:** Code compilation must occur on a central coordinator and cannot occur client-side due to the sandboxed nature of WebAssembly. In order for submitters to compile code in their browser, they would require a copy of the compiler (compiled to WebAssembly) and all libraries to be available locally. This would require at least 2 GB of data to be downloaded and to be loaded into a contiguous section of the virtual address space whenever the submission webpage is visited. While there are current attempts to run Clang on the web via WebAssembly, the task of porting a C++ compiler to WebAssembly is out of the scope of this project [4].

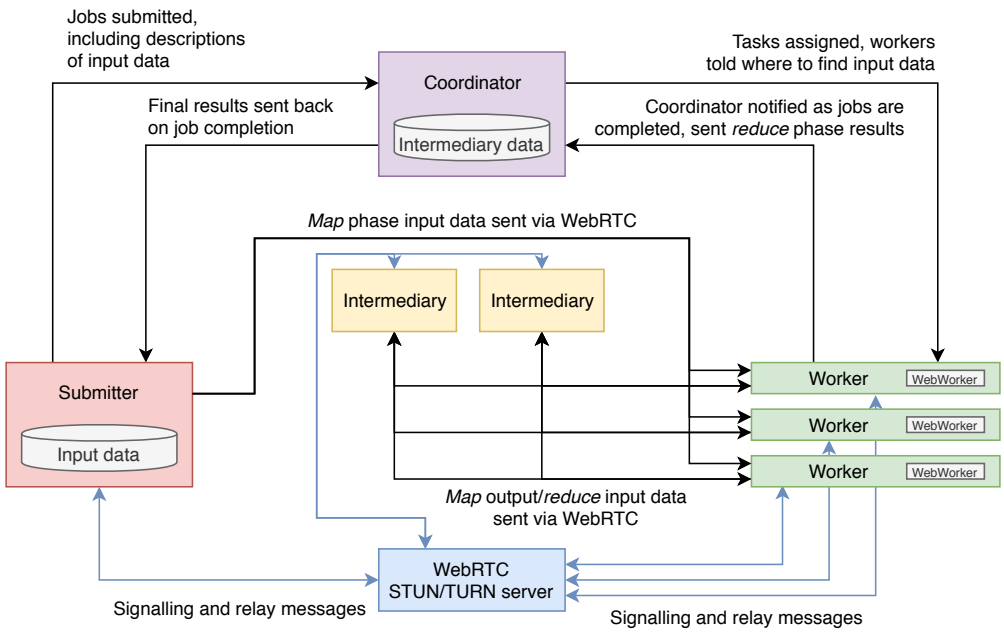

Figure 2: WebMesh architecture overview.

Therefore, all input code is compiled by executing the Emscripten `emcc` binary on the coordinator server, which works similarly to LLVM but compiles from C++ source code to a JavaScript file containing needed imports, syscall implementations, and more, as well as a WebAssembly binary.

Important to note, the `-Os` optimization flag is used as this indicates to the Emscripten toolchain that it should optimize for size. While potentially not the most computationally-efficient compilation option, this reduces code size of both the JavaScript and WebAssembly files, reducing load on the coordinator, reducing download times, and decreasing the memory footprint of each worker. This compilation flag decreased code size by approximately 7% in testing as compared with `-O3`, a significant savings when multiplied across many workers.

- **Job scheduling:** WebMesh uses a simple FIFO scheduler, first assigning jobs to idle worker nodes, then assigning sets of jobs one worker at a time until each worker has a fixed local queue of jobs to complete (discussed later). Since all of the jobs for each phase within an overarching task are created and assigned all at once, this scheduler ensures locality for repeated jobs that rely on the same source code. This means that workers can reuse instantiated WebAssembly modules on workers and can reuse connections between workers and submitters for downloading input data.

  Jobs are given a configurable timeout, which begins as soon as the job is assigned to the worker. Jobs are then retried on different workers (if available) if they fail to be completed within that timeout.

- **WebRTC signalling:** The central coordinator helps workers, intermediary nodes, and submitters negotiate WebRTC connections through the exchange of session descriptions and Interactive Connectivity Establishment (ICE) candidates. See Figure 3 for the detailed process.

  In order to decrease the setup time for WebRTC connections, the coordinator will hold a few dozen connection stubs from each submitter and intermediary node, so that once workers begin connecting, a portion of the setup process is already completed. This also speeds up the ICE candidate exchange because in Google Chrome, the WebRTC endpoint on the worker begins

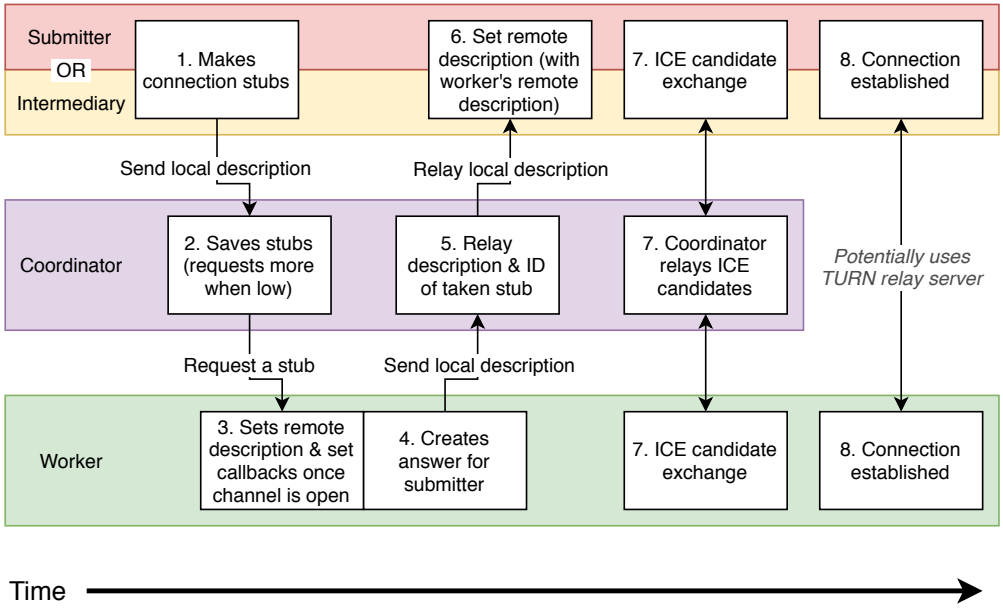

Figure 3: WebRTC connection establishment.

using the STUN server to build ICE candidates as soon as the local RTC channel is created.

The TURN server is used after connection establishment in the case that, as determined in the ICE candidate exchange, a relay is the only viable option due to network conditions. The combined STUN and TURN server is a third-party package available for free on Github [9].

- **Intermediary data:** The coordinator may hold the intermediary data between phases in memory, if there are not enough intermediary nodes connected. In effect, for the MapReduce implementation, this means that the shuffle phase occurs during the map phase, as results begin being sorted as they are received. The use of intermediary nodes is discussed in Section 3.3, and they are preferred to using the in-memory coordinator shuffle, as evidenced by the results in Section 4.3.

3.2 Workers

Workers in the system serve to execute many short tasks. To reduce the time spent waiting between tasks, workers have a local job queue. This eliminates the need to wait for the coordinator to process a completed job and re-assign a new job each time a job is completed. Given that an individual job may only take milliseconds to execute, but latency can be 100 milliseconds or more, the local queue is a massive efficiency advantage despite drawbacks of assigning too many tasks to a worker that may be unreliable. The local queue is capped to ensure that workers are assigned jobs at a rate proportional to the speed at which they complete them.

WebMesh workers take advantage of Web Workers, ensuring that WebAssembly execution is non-blocking to the main event loop. For workers, the main event loop is responsible for communication with the coordinator as well as data requests and WebRTC communication with submitters/intermediaries, and blocking it would prevent new jobs and new data from being received. Web Workers utilize message passing for synchronization – the Web Worker for each client waits to receive a job with all of its input data, executes the job, and sends a message back to the main event loop. On receipt of this message, the next job is sent to the Web Worker.

The Web Worker keeps a reference to the most-recently-used WebAssembly module and if the next job uses the same source code as the previous job, the downloaded and instantiated WebAssembly module can be completely reused, vastly decreasing overhead

and server load. Furthermore, due to the parallelism given by Web Workers, queued jobs are able to request input data from the submitters while other jobs are executing.

3.3 Intermediary nodes

Intermediary nodes are similar to workers, but instead of executing WebAssembly code, they act as a data store in between execution phases. In MapReduce, WebMesh intermediary nodes manage the shuffle phase by receiving results from workers completing map jobs, and sending data to workers requesting input data for reduce jobs.

When using intermediary nodes, workers running map tasks are notified about all nodes selected as intermediary nodes for the given task. Workers then hash the keys of the map output data to deterministicly divide it among all of the intermediary nodes, thus making each intermediary node responsible for a defined subset of the keys.

The intermediary nodes then divide their subset of keys further into a pre-configured number of buckets, and each reduce job is associated with a specific bucket of one of the intermediary nodes. Workers executing a reduce task will contact a single intermediary node and request one bucket of keys and values at a time upon which to execute reduce tasks. Therefore, the coordinator never needs to know the number of keys or amount of output data in order to assign reduce jobs.

Figure 4 helps to explain the benefits of using intermediary nodes. On the left, (a) depicts WebMesh operation without intermediary nodes – mappers and reducers send and receive data using the coordinator as a clearinghouse. While this requires the least number of connections, only $m + n$ where m is the number of mappers and n is the number of reducers, this limits the size of the shuffle phase to only what the coordinator can hold in memory, or in some ephemeral database. This also create a bottleneck of having only one node executing the shuffle phase of MapReduce.

In the middle, (b) shows the worst case scenario if shuffling were done completely decentralized by passing data directly from mappers to the reducer responsible for each key. Each mapper may have a random sample of the input data, and thus each mapper could have values for every key, thus requiring a complete set of $m*n$ connections between all mappers and reducers.

Finally, (c) shows the optimization provided by having a few intermediary nodes execute the shuffle phase. Mappers may need to contact all intermediary nodes, and when executing many reduce tasks, reducers may need to contact all intermediary nodes. Still, this only creates $m * i + n * i = i * (m + n)$ connections, where i is the number of intermediary nodes. Given that i is always much less than m or n, this is much more efficient than the highly-decentralized approach seen in (b), yet still provides many of the benefits of the decentralized shuffle.

3.4 Execution model

While the framework could be easily adapted for other uses, the implemented execution model of WebMesh is based on MapReduce [18]. Users provide a `map()` and `reduce()` function in C++, a set of input files, and submit their job to be run. The input files provided are used for the map phase, and sent directly to the workers running jobs in that phase. Workers return the results to either the coordinator or to intermediary nodes, which send out the input data for the reduce phase to workers again. Workers send back those results to the coordinator, and once all results are submitted, the aggregate is sent to the original submitter, who can view or download the results as a file. This flow is shown in Figure 5.

With slight modification, WebMesh is easily adaptable to a one-phase model or some polyphasic computing models. Phases are defined by a state machine within the coordinator, and thus are easily editable for different computing models. For workers, the data source is specified as either coming from another node, with the connection stub given, or as coming from the coordinator, with the data provided in the job description. MapReduce was selected for this evaluation due to its ease of understanding as a multi-stage execution model, and due to its well-known proof-of-concept use cases such as word count.

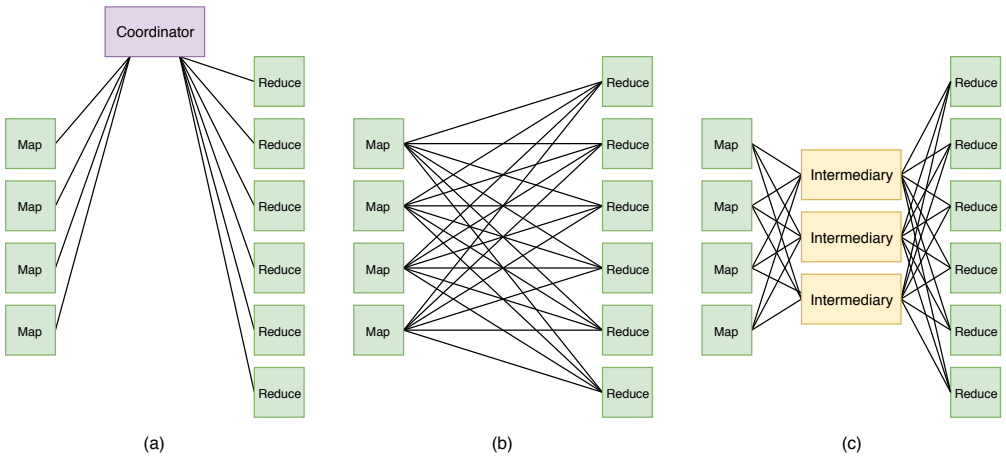

Figure 4: Comparison of methods to connect map output data to reduce input data. First, (a) shows WebMesh operation using the coordinator for an in-memory shuffle. Second, (b) shows a worst-case scenario for decentralized operation in which all mappers have entries for all keys, and thus require a complete set of connections between all mappers and all reducers. Third, (c) shows the efficiency gain of having intermediary shuffle nodes.

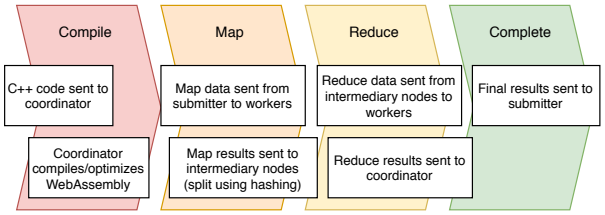

Figure 5: MapReduce model as implemented in WebMesh.

```
vector<pair<string, MVT>> map(const string
    doc);
pair<string, RVT> reduce(const string key,
    vector<MVT> vals);
```

Figure 6: WebMesh C++ API.

3.5 Programming model

As mentioned, the user must implement a `map()` and `reduce()` function. These are then compiled into a larger MapReduce template based loosely off of a Google proposal for standardized MapReduce types in C++ [30]. For simplicity in interoperability with JavaScript types, the MapReduce keys must always be of type `std::string`, although the value types may take on any type – even classes such as a `std::pair`. The API can be seen in Figure 6, with `MVT` being the map value type, and `RVT` being the reduce value type. Based on the types given in the function declaration, the coordinator dynamically inserts Emscripten's Embind bindings when compiling the WebAssembly module. Two example API usages for word count and matrix multiplication tasks can be seen in Figure 7.

```
vector<pair<string, int>> map(const string
    doc);
pair<string, int> reduce(const string key,
    vector<int> vals);
```

```
vector<pair<string, pair<int, float>>> map
   (const string mat);
pair<string, float> reduce(const string
   key, vector<pair<int, float>> vals);
```

Figure 7: Word count (top) and matrix multiplication (bottom) function declarations using the WebMesh C++ API.

4 Evaluation and Discussion

4.1 Methodology

Performance testing was run with the coordinator on an AWS `t2.micro` instance, and the STUN/TURN server running on a separate AWS `t2.micro` instance. At the time of writing, these instances have 1 "vCPU" and 1 GB of memory.

The submitter was run on a wireless-networked mid-2015 MacBook Pro running macOS 10.14.3 with an Intel Core i7 processor, 16 GB of memory, and input files stored on a solid state drive. The submitter was primarily evaluated using Google Chrome version 73.0.3683.86.

Workers and intermediary nodes were run on the same internal university network on wired Linux desktops with Intel Core i5 processors and 8 GB of memory. Workers were primarily evaluated using Google Chrome versions 73.0.3683.86 and 73.0.3683.103, although workers were run using Chrome's headless (non-UI) option.

When intermediary nodes were used, three intermediary nodes were run with 10,000 hash buckets per node for the reduce phase, splitting the data into 30,000 reduce jobs.

Workers were also tested using AWS Lambda instances running Headless Chrome via an implementation of the `serverless-chrome` project [12], which was configured to wait on the WebMesh worker webpage for 25 seconds and then time out. Using Lambda for an execution environment presented challenges, as AWS Lambda will run multiple Lambdas for only one request to anticipate future load. This made it hard to measure the number of concurrent workers at any given time. Therefore, Lambda workers were not used for performance testing but for testing unreliable workers – workers would rapidly disconnect and reconnect due to the aforementioned design of AWS Lambda.

The primary test use case was the canonical MapReduce word count job. The test data used was a subset of the Yelp dataset [16]. The dataset includes a 5.35 GB JSON file containing reviews from Yelp.com, although the JSON formatting was irrelevant for the purposes of the evaluation. The dataset was processed to be split into file chunks of a variety of sizes: 5 KB, 50 KB, and 500 KB.

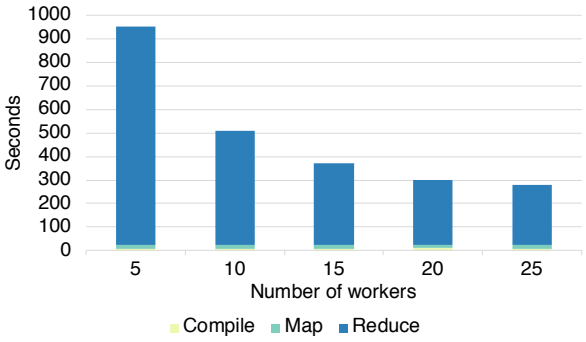

Figure 8: Time, by phase, to complete word count job on 100 500 KB files with an in-memory shuffle on coordinator, as affected by number of workers

4.2 Word Count with coordinator shuffle

Figure 8 shows the total time taken to complete a word count job on 50 MB from the Yelp dataset, when split into 100 500 KB files and using the coordinator for an in-memory shuffle. It is possible to observe significant efficiency gains from adding up to 25 workers. The scaling is quite optimal, going from 929 seconds with five workers to 349 seconds with 15 workers, incurring only a 12.7% overhead with three times more workers. This is due to the distribution of over 244,000 reduce jobs in this example.

While the map jobs take a very small fraction of the total time, there is still little time benefit in the map phase to adding more workers. In fact, more workers made the map phase take slightly longer, changing from 13 seconds with five workers to 14.5 seconds with 25 workers. This likely has to do with data transfer, module instantiation, and WebRTC connection overhead. The first time a worker receives a map or reduce job for a given overall submission, it must make two requests to download a JavaScript and WebAssembly binary, allocate memory for the WebAssembly module, and initialize it. This process can take a few seconds, and in a case where there are not many map jobs as compared to reduce jobs (hundreds versus hundreds of thousands), this overhead is quite substantial. In addition, there is overhead to establishing a WebRTC connection, which is maintained for the lifetime of each worker. In the case of more workers, the WebRTC connection overhead is more prevalent because the coordinator must relay each worker's connection information and

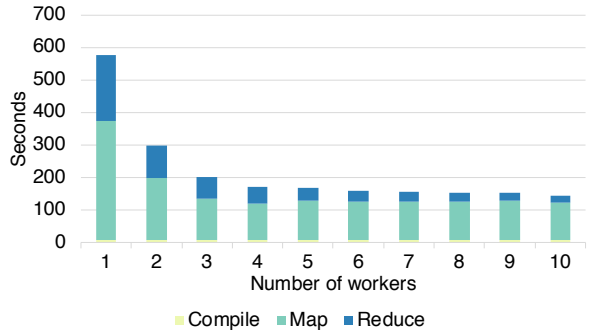

Figure 9: Time, by phase, to complete word count job on 2000 500 KB files (1 GB total) with shuffling on three intermediary nodes, as affected by number of workers

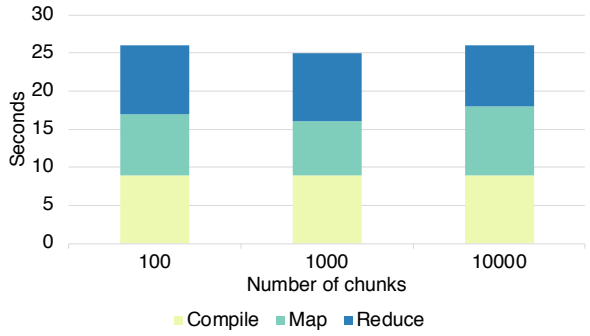

Figure 10: Time, by phase, to complete word count job on 50 MB on ten workers and three intermediary nodes, as affected by chunk size (100 chunks = 500 KB chunks, 1000 chunks = 50 KB chunks, 10000 chunks = 5 KB chunks)

ICE candidates at nearly the same time when jobs are assigned.

4.3 Word Count with intermediary shuffle

Using intermediary nodes for the shuffle phase allows processing much larger datasets. Figure 9 shows the time taken to run the word count job on 1 GB of data from the Yelp dataset, broken into 2,000 500 KB files. When using intermediary nodes with such a large amount of map data, scaling can be observed in both the map and reduce phases. When going from one worker to three workers, the time taken in the map phase decreases from 364 seconds to 126 seconds (3.8% overhead), and the time in the reduce phase decreases from 203 seconds to 65 seconds (-3.9% overhead).

These 1 GB jobs in Figure 9 completed in an even faster time than the 50 MB jobs in Figure 8, highlighting the benefits of the intermediary nodes and use of WebRTC in WebMesh. In fact, despite overhead from job scheduling, connection establishment, compiling, and more, these jobs completed faster on WebMesh than locally. It took over eight minutes (527 seconds) to run and print word count results for identical data in an analogous C++ program on a 2015 MacBook Pro, while the job on WebMesh with two workers and three shufflers completed in just under five minutes (299 seconds). After adding up to ten workers, the job took 146 seconds on WebMesh, 3.6 times faster than running the same job locally.

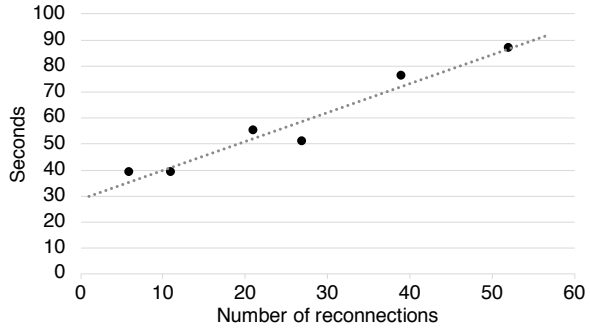

Figure 11: Total time to complete a word count job using AWS Lambda on 10 500 KB files, with three concurrent AWS Lambda invocations and an in-memory shuffle on the coordinator, as affected by the number of measured reconnections. (slope = 1.1103, R^2 = 0.9483)

Figure 10 shows the time taken to complete a word count job on 50 MB from the Yelp dataset using ten workers and three intermediary nodes. In this figure, the 50 MB have been split into a different number of chunks: 100 chunks (500 KB each), 1000 chunks (50 KB each), and 10000 chunks (5 KB each). The time for the job is nearly identical regardless of the chunk size, indicating that WebMesh users do not need to concern themselves with finely optimizing input data to fit system constraints for efficiency.

4.4 Unreliable Workers

AWS Lambda proved to be an unreliable environment in which to run workers. While Lambda is an event-based service, it attempts to preemptively stop and start Lambdas to ensure quick scaling and constant availability. This means that workers run for an unpredictable number of times, with an unpredictable number of reconnections. Figure 11 shows the total time to complete a word count task on the Yelp dataset for 10 500 KB files with three simultaneous AWS Lambda invocations and an in-memory shuffle on the coordinator. The time increases with the number of reconnections, or failures, because jobs time out and new workers need to instantiate WebAssembly modules, download input data, and more.

The trend of increasing time based on the number of reconnections follows a generally linear pattern with a low slope (slope = 1.1103, R^2 = 0.9483), which indicates a fairly efficient system in the face of unreliable workers. It is expected that in a real-world use case environment, workers will be quite unreliable as it is very easy to close a browser tab and cut off execution. Furthermore, if running in the background, some browsers will kill execution of a tab or throttle them to increase performance for active tabs, thus increasing the need for an efficient system in the face of failures.

4.5 Versatility

As seen in Figure 6 showing the API, there is support for a variety of use cases without any changes needed outside of the submitted source code. A naive implementation of matrix multiplication yielded correct results when tested with large square matrices. This implementation used the matrix multiplication function declarations seen in Figure 7 without any additional modification to support the `std::pair` type in C++.

Additionally, the system is versatile due to its high portability. In addition to the testing done on the Headless Chrome Linux machines and on AWS Lambda, WebMesh workers have been run on smartphones and a variety of desktop environments. Any device supporting Chromium is a valid WebMesh worker, and as more browsers conform to the specifications for new technologies like WebRTC and WebAssembly, even more devices will be valid WebMesh workers. In the near future, even cars will support Chromium and will be able to be WebMesh workers [29].

4.6 Cost

To break down potential cost savings by using idle compute time on web browsers in place of a serverless provider such as AWS Lambda, an analogy can be made between WebMesh workers and AWS Lambda requests. For the sake of simplicity, assume a WebMesh worker has equivalent power to an AWS Lambda invocation. Ignoring the AWS "free tier," Lambda charges $0.0000002 per request, and $0.00001667 per GB-second for Lambda [10].

If a WebMesh worker were to be billed at the rate of a Lambda invocation, with one WebMesh worker being treated the same as a Lambda worker with 128 MB of memory, the cost of running a job using 1000 workers for five minutes would be approximately $0.625, not counting costs of the coordinator or STUN/TURN servers. Assuming 10% of revenue goes to client workers (due to replication, overhead, and network costs), each worker would receive up to $0.00075 per hour, or $0.54 per month.

If one were to run a Bitcoin miner in their browser, achieving hash rates of up to 5 MH/s with a network hash rate of approximately 45.8 GH/s, Bitcoin to USD conversion of 1 BTC = $5223.93, and block reward of 12.5 BTC, the expected revenue would be $0.00003082 per month [2]. While imprecise and subject to inflationary claims about Bitcoin, the fact that Bitcoin mining is popular yet has an expected return four orders of magnitude less than the expected return from being a WebMesh worker indicates that there is untapped opportunity in leveraging idle computational power. If revenue sharing could be increased to greater than the assumed 10%, or WebMesh is able to charge even more AWS Lambda due to a better feature set from its peer-to-peer WebRTC capabilities, then the upside increases even more for potential workers.

4.7 Shortcomings

This work also exposed several potential flaws and issues in implementing a web-based computational

framework such as WebMesh.

The greatest issue is the reliance on a TURN server to ensure reachability, as that eliminates the cost savings from using WebRTC for communication and creates a significant bottleneck for all communication. There are no clear solutions to this besides attempting to load balance work away from nodes which are behind a restrictive firewall or NAT, to minimize costs and bottlenecks incurred by using the TURN server. This is not a very reasonable step, as most situations in which WebMesh was tested used enterprise-grade networking, and had to use the TURN server except in cases where workers were on the same machine as the submitter. Perhaps on personal networks, the TURN server is less likely to be needed.

Another issue experienced throughout the development of this project was the volatility of new technologies such as WebAssembly and WebRTC. Features that worked previously in Apple's Safari browser broke with changes to WebRTC, and an update to WebAssembly forced changes in the WebMesh worker code. These changes, however, are welcome. WebAssembly is nowhere near complete, only recently having launched its minimum viable product, and does not have sufficient support for many critical features such as dynamic linking.

As discussed earlier, the sandboxed nature of WebAssembly prevents submitters from locally compiling C++ code to WebAssembly, thus requiring a central coordinator to run compilation on behalf of all submitters. This is easily exploitable through C++ templates, which are often used to shift work to be done at compile time and may even be Turing complete [34]. A malicious actor may attempt to use template metaprogramming in C++ as a way to prevent incurring any cost of worker time, or to overwhelm the coordinator with compilation tasks. A simple solution to this template risk is a timeout on compilation, which is what existing C++ to WebAssembly compilation servers do. However, this is just one example of a vulnerability created by having server-side compilation.

Another security risk to consider is that workers will be executing arbitrary code from a potentially untrusted source. However, as mentioned previously, WebAssembly provides the same security and sandboxing guarantees as current JavaScript VMs in web browsers. While browsers are still subject to many security exploits, the greater risk of WebMesh is the untrusted source code. On a typical commercial website, even if browser exploits exist, there is an implicit agreement between the company managing the site and the users that user security will not be compromised by browsing the site. While out of the scope of this project, a real-world implementation of WebMesh would also likely require sharding, "networks of trust," or identity verification for people submitting tasks to ensure legal compliance and to prevent malicious programs from running on the network.

Another issue in WebMesh is the trade-off of choosing the in-memory coordinator shuffle versus running shuffle on intermediary nodes. The in-memory coordinator shuffle does not scale very well and presented a bottleneck even during testing. When attempting to run a naive matrix multiplication algorithm on square matrices, there were n^3 values emitted as a result of the map phase (n is the length of a side of an input matrix). The number of intermediary values overwhelmed the coordinator in cases of large matrices. However, when tested using intermediary nodes, the square matrix problem divided well across the nodes and was more scalable.

Yet, using intermediary nodes presented challenges when attempting to test on a greater number of workers, due to WebRTC limitations. Browsers impose limitations on the number of simultaneous WebRTC connections, and Chrome has imposed limits ranging from 10-256 simultaneous peer connections at once. When testing, we experienced browser errors with as low as 15 workers and 5 intermediary nodes. This occurred not only due to the number of WebRTC connections, but also the burden put on the signalling server (coordinator) and the STUN/TURN server. This means that there is still a limit to scalability, although this may be resolved as WebRTC becomes an increasingly-used and more well-supported technology.

4.8 Future work

While we used MapReduce as a proof-of-concept implementation on top of WebMesh, it is flexible to other computing models. Ideally the WebMesh API itself would be generalized to support a wider range

of distributed operators on data in any order.

Given a more flexible system, WebMesh may perform even better on jobs that are less data-intensive than MapReduce jobs. For example, focusing on a compute-intensive task, WebMesh would be able to equally distribute tasks among workers to reap the benefits of distributed execution, without incurring the overhead of data transfer. MapReduce in particular is difficult due to the data-intensive shuffle phase, and future work should explore running modern serverless applications, which could be less data-intensive, on a similar framework.

As mentioned above, exploring offline compilation is another opportunity. Perhaps if some trusted workers were able to download a WebAssembly copy of LLVM, those workers could be dedicated compilers on behalf of the rest of the network. This would allow the submitter to send the raw C++ code to those compiler workers as part of the job startup process. Then, those compiler workers would receive WebRTC connections from workers executing the tasks, therefore offloading much of the work from the coordinator.

Another unexplored avenue is trust and replication of jobs. There is a significant body of research on trust in volunteer and grid computing. Using this research could allow submitters to have more confidence in the results they receive. Even if not replicating jobs for trust reasons, it would be helpful to build in job replication for efficiency. Replication would reduce the need to wait for jobs to time out, to wait for stragglers to finish, or to re-assign jobs to new workers if the old ones disconnect.

There is also opportunity in creating a JavaScript-based network filesystem to provide a virtual filesystem within the C++ programs running on WebMesh. Emscripten already provides a fully-implemented virtual filesystem within its JavaScript "glue code" and WebAssembly imports, and replacing that with a network filesystem could allow programs better means of communicating in a more complicated framework besides only message-passing between jobs.

Future work may also include a thorough investigation of the security impacts of web-based distributed frameworks, particularly with respect to WebAssembly and WebRTC. Investigation into more social-oriented security, such as methods of trust and reputation-sharing, may provide interesting results.

Finally, future work could include further research into the economic viability of such a system. While Section 4.6 showed that there may be significant economic opportunity for worker nodes, this analysis was done without considering the replication, trust, and security ramifications of this system in the real world.

5 Related Work

Many systems exist for volunteer and serverless computing, although WebMesh is the first to have used WebAssembly to allow portable C++ code execution and WebRTC to allow peer-to-peer networking over the web.

Several systems exist which expose a purely JavaScript API to allow for distributed computing in the browser [19]. MRJS (MapReduce JavaScript) is the most similar of these, providing the same MapReduce framework in plain JavaScript with dynamically-submitted tasks [32]. QMachine provides a more general framework in JavaScript with a custom execution model and API [35]. Sashimi, deThread, and Ar are also similar frameworks that provide capability for distributed computing in JavaScript [28][7][33]. Gray Computing and BoomerangJS are middleware that allow pre-determined tasks to be run in a distributed manner using JavaScript [31][3]. Despite all of these systems and frameworks, WebMesh is the first to use WebAssembly and the first to use WebRTC in running tasks across volunteer workers.

There are also startups operating in a similar field. Mentioned previously, Amazon and Google both offer serverless computing products [1][13]. While focused more on event-driven functions instead of building out distributed frameworks, Google has increasingly developed into this field and has just announced Google Cloud Run, a "serverless execution environment" for container-based services, marking another step in the direction of frameworks for data processing and computation [25].

Computes, Inc., recently acquired by Magic Leap, had a project in development to create a monetized, distributed, serverless computing platform [5][6]. This explored a field similar to WebMesh, although

expected users to install a browser supporting the IPFS protocol, increasing the barrier to entry. Furthermore, the product never launched before the company was acquired.

There are many companies which also offer services using idle browser time, or running in the background of webpages, to mine cryptocurrency. While not allowing dynamic tasks to run, this has the same fundamental principle of having unreliable and flexible worker nodes as WebMesh. One such prominent company is MinePrize [11].

Pywren is a framework meant to run general computation on top of serverless providers such as AWS Lambda [22]. Pywren provides a potential API that WebMesh could conform to in order to provide more a more flexible computational model which combines a wide set distributed operators.

Finally, MOON: MapReduce on Opportunistic eNvironments is an extension of Hadoop, adapted to run on unreliable volunteer environments similar to that of WebMesh workers [24]. MOON represents an unexplored area of this project, presenting an opportunity to incorporate concepts from their work to create a more reliable service within WebMesh.

Finally, blockchains such as Ethereum also provide a distributed computing model which pays users to execute code through so-called "smart contracts." Yet, this does not allow for large computing tasks to be done, as any code executed is run on all machines connected to the Ethereum blockchain, and billed at a very high rate [17]. In contrast, WebMesh is focused on running many high-computation and/or high-data tasks.

6 Conclusion

In this paper we presented WebMesh, a browser-based computational framework for serverless applications. We found that WebMesh is a scalable system, that it is able to provide computation 3.6 times faster on word count jobs than running locally, and that it is able to handle 200,000+ concurrent jobs. Furthermore, we presented the versatility of the system in its execution and programming model. We also showed its resilience despite unreliable workers. We showed that such a system functions reasonably and may be economically viable, creating opportunity and value out of preexisting and unused resources.

Due to resource limitations and WebRTC limitations, we could not test scalability with very high numbers of workers or a high diversity of worker nodes. Future work should explore the security and trust ramifications of WebMesh, as well as the ability to remove bottlenecks such as the TURN server. In the long term, interesting future work would study the ability to have a flexible computation model that goes beyond what MapReduce may offer.

7 Acknowledgements

I would like to thank Rodrigo Fonseca for providing guidance on this project and teaching me so much about networks and distributed systems during my time at Brown. Thank you to Tom Doeppner for taking the time to be my reader and Shriram Krishnamurthi for your advice throughout undergrad. In addition, thank you to all of my mentors and thank you to everyone who spent the time reading and editing this paper.

In general, I would like to thank my friends for always being there and continuously teaching me new things. Thank you in particular to Kat for always laughing at my jokes.

I also would like to thank my parents, Bob and Rita, for their constant support and love. You made me into the person I am and created opportunities for me that you never had. I will never forget all the sacrifices you have made.

References

[1] AWS Lambda. `https://aws.amazon.com/lambda/`. Accessed: 2019-04-05.

[2] Bitcoin mining profitability calculator. `https://www.cryptocompare.com/mining/calculator/btc`. Accessed: 2019-04-05.

[3] BoomerangJS. `http://www.boomerangjs.org`. Accessed: 2018-09-14.

[4] Clang In Browser (cib). `https://github.com/tbfleming/cib`. Accessed: 2019-04-05.

[5] Computes. https://computes.io. Accessed: 2018-09-14.

[6] Computes, Inc. Joins the Magic Leap Family. https://www.magicleap.com/news/press-release/computes-joins-magic-leap. Accessed: 2019-04-07.

[7] deThread. https://github.com/deThread/dethread. Accessed: 2018-09-14.

[8] Emscripten. https://emscripten.org. Accessed: 2018-03-15.

[9] Free open source implementation of turn and stun server. https://github.com/coturn/coturn. Accessed: 2019-04-05.

[10] Lambda pricing details. https://aws.amazon.com/lambda/pricing/. Accessed: 2019-04-05.

[11] MinePrize. https://mineprize.com/#work. Accessed: 2018-09-14.

[12] serverless-chrome. https://github.com/adieuadieu/serverless-chrome. Accessed: 2019-04-05.

[13] Serverless Computing | Google Cloud. https://cloud.google.com/serverless/. Accessed: 2019-04-05.

[14] Using Web Workers: Web Workers API. https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API/Using_web_workers. Accessed: 2018-04-06.

[15] WebRTC. https://webrtc.org. Accessed: 2018-10-02.

[16] Yelp open dataset. https://www.yelp.com/dataset. Accessed: 2019-04-05.

[17] Buterin, V. *A Next Generation Smart Contract & Decentralized Application Platform*, 2014.

[18] Dean, J., and Ghemawat, S. Mapreduce: Simplified data processing on large clusters. In *6th Symposium on Operating System Design and Implementation (OSDI 2004), San Francisco, California, USA, December 6-8, 2004* (2004), pp. 137–150.

[19] Fabisiak, T., and Danilecki, A. Browser-based harnessing of voluntary computational power. *Foundations of Computing and Decision Sciences 42*, 1 (2017), 3 – 42.

[20] Haas, A., Rossberg, A., Schuff, D. L., Titzer, B. L., Holman, M., Gohman, D., Wagner, L., Zakai, A., and Bastien, J. F. Bringing the web up to speed with webassembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017* (2017), pp. 185–200.

[21] Jangda, A., Powers, B., Guha, A., and Berger, E. Mind the gap: Analyzing the performance of webassembly vs. native code. *CoRR abs/1901.09056* (2019).

[22] Jonas, E., Venkataraman, S., Stoica, I., and Recht, B. Occupy the cloud: Distributed computing for the 99%. *CoRR abs/1702.04024* (2017).

[23] Lardinois, F. Google says there are now 2 billion active chrome installs, Nov 2016.

[24] Lin, H., Ma, X., Archuleta, J. S., Feng, W., Gardner, M. K., and Zhang, Z. MOON: mapreduce on opportunistic environments. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing, HPDC 2010, Chicago, Illinois, USA, June 21-25, 2010* (2010), pp. 95–106.

[25] Manor, E., and Teich, O. Announcing Cloud Run, the newest member of our serverless compute stack.

[26] Matthews, P., Rosenberg, J., and Mahy, R. Traversal Using Relays around NAT (TURN): Relay Extensions to Session Traversal Utilities for NAT (STUN). Tech. Rep. 5766, Apr. 2010.

[27] Matthews, P., Rosenberg, J., Wing, D., and Mahy, R. Session Traversal Utilities for NAT (STUN). RFC 5389, October 2008.

[28] Miura, K., and Harada, T. Implementation of a practical distributed calculation system with browsers and javascript, and application to distributed deep learning. *CoRR abs/1503.05743* (2015).

[29] Musk, E. About to be upgraded to chromium. Twitter post, March 2019. Accessed: 2019-04-05.

[30] Mysen, C., Crowl, L., and Berkan, A. C++ Mapreduce. Tech. Rep. 3563, March 2013.

[31] Pan, Y. *Gray Computing: A Framework for Distributed Computing with Web Browsers*. PhD thesis, Vanderbilt University, 2017.

[32] Ryza, S., and Wall, T. MRJS: A javascript mapreduce framework for web browsers.

[33] Vasilakis, N., Goel, P., Demoulin, H. M., and Smith, J. M. The web as a distributed computing platform. In *Proceedings of the 1st International Workshop on Edge Systems, Analytics and Networking* (New York, NY, USA, 2018), EdgeSys'18, ACM, pp. 7–12.

[34] Veldhuizen, T. L. C++ templates are turing complete.

[35] Wilkinson, S. R., and Almeida, J. S. Qmachine: commodity supercomputing in web browsers. *BMC Bioinformatics 15* (2014), 176.