# Arke: An in-browser implementation of MapReduce

James Sedgwick Peter Wilmot Stephen Poletto

Final Project - CSCI 2950-u - Fall 2011

**Abstract**

This paper describes Arke, a JavaScript implementation of Google's MapReduce. Rather than running on a fixed cluster in a data center like most distributed computational frameworks, Arke relies on the computing power of individual browsers across the open Internet. Volunteers connect to Arke's web application, offering their web browser engine as a worker in the system. Further, Arke is a service; users may upload jobs and data conforming to a simple specification in order to delegate their computation to a dynamic cloud of volunteer nodes.

## 1 Introduction

With greater availability of cloud computing services, more and more academics, businesses and individuals are offloading their data-intensive and computationally-expensive jobs to private clusters of machines. We believe there is an opportunity to build a crowd-driven alternative to these services, with an intuitive and easy-to-use application that lowers the barrier to entry for job providers. Arke aims to seize this opportunity by providing a scalable web service in which users can both upload their own jobs and volunteer their browsers as workers in the system. Existing services such as SETI@Home and Folding@Home have demonstrated that a volunteer-based computational model is indeed feasible, and that users are willing to contribute their computational resources to solving hard problems. However, these services are problem-specific and lack extensibility. They also require their users to download and install applications in order to join their computational networks.

Arke aims to solve these problems by utilizing the browser. Rather than requiring installation of additional applications or plugins, Arke depends upon the native javascript execution environment of the user's browser. Volunteers connect to the Arke server to fetch small chunks of data and functions to perform with the data. They then send their results back to the server when completed. With the introduction of HTML 5 WebWorkers, it's possible to perform computationally intensive work in the browser in a separate thread, keeping the browsing experience smooth and responsive for the user.

Arke is built to be a general-purpose computational tool. It utilizes the MapReduce model, in which users specify a map function, a reduce function, and an input data set. We believe MapReduce is expressive enough to model a range of computational problems across a range of disciplines. Further, MapReduce has proven to be useful in a variety of contexts, as shown by the success of Apache Hadoop. Arke is designed primarily for low-data, high-computation jobs, such as NP-complete problems, raytrace rendering and Monte Carlo simulations. Since all data must be communicated to client nodes over the open Internet, problems with high computation per byte are best suited to the Arke service.

In this paper, we present the design and implementation of Arke, as well as an evaluation of its performance. Section 2 provides a background of the MapReduce model as well as the technologies employed in our implementation. Section 3 describes the architecture of Arke, outlining the client-side logic and server design, including a description of our backing datastore. Section 4 describes our testing infrastructure, explaining how we conducted our experiments for evaluating the scalability and usefulness of the system. Section 5 presents and describes our results. Section 6 details ideas for future improvements. Finally, section 7 concludes with some closing remarks.

# 2 Background

## 2.1 MapReduce

MapReduce is a distributed computational framework created by Google to parallelize computation on large data sets across large clusters of commodity hardware. MapReduce was designed to focus on fault tolerance and scalability. Since its introduction in 2004, MapReduce has spawned an open source equivalent called Hadoop, which in turn has spawned numerous offshoot frameworks. Many large companies rely on MapReduce for their big data needs. Many parallelizable problems are easily cast into MapReduce's straightforward computational model. Developers specify a map function to operate on chunks of the input data. The map function produces intermediate key-value pairs, which are then grouped by key and passed to a reducer function. The reducer receives all values for a given intermediate key. We chose to replicate MapReduce's computational model because of the model's applicability and familiarity to potential Arke users.

## 2.2 Node.js

Node.js is a JavaScript framework for writing highly scalable web applications. It is built on top of Google's V8 JavaScript engine. Node utilizes event-driven, asynchronous I/O to handle many concurrent requests by minimizing overhead and competition for resources. Node.js thus stands in stark contrast to web servers like Apache, which have a high level of overhead per connection. As a result, we felt Node.js was well suited to Arke's needs — simultaneously serving many persistent connections that require significant concurrent I/O without blocking other requests. Additionally, there are excellent third party libraries for Node.js that make the use of RPCs and web sockets — two key components of our design — effortless. Node is lightweight, and given Arke's near-stateless job server design, we can accomplish cheap scalability by spinning up additional Node instances on commodity machines behind a load balancer if necessary.

## 2.3 MJRS

Arke is very similar in its design goals to MJRS, an in-browser implementation of MapReduce implemented by Ryza and Wall in 2010. Arke focuses on three major areas of improvement over MJRS.

### 2.3.1 Scalability

The authors of MJRS note that the job server became a bottleneck in their system after about 80 simultaneous connections. They attribute the bottleneck to running their server on top of Apache. Apache and other conventional web servers spawn a fresh process for each client connection, and these processes block each other, thus limiting its performance with many concurrent connections. We hoped to handle a larger number of simultaneous connections and I/O operations by exploiting Node's event-driven runtime and non-blocking I/O.

### 2.3.2 Fault Tolerance

The MJRS authors did not account for client disconnects. In order for Arke to be usable as a full-featured web service, fault tolerance needs to be handled intelligently. If a client leaves the page or disconnects for any other reason, the work unit they were working on needs to be re-assigned to another worker.

#### 2.3.3 Software as a Service (SaaS)

While MJRS provided a demonstration of the feasibility of in-browser MapReduce, it is not accessible to external users, nor is it clear how to use it for a one-off job. We wanted to build Arke such that it could be deployed to real world end-users after conducting our own experiments with it. We also decided to open-source the codebase, so that others can build on top of the Arke framework or create their own platform using its engine.

## 3 Architecture

### 3.1 Client

Upon connecting to the Arke web application, a user's browser establishes a full duplex connection with the job server. We utilize Now.js, a Node module that provides a simple interface to call remote procedures on both client and server in real time. Now.js is built on top of Socket.io, a JavaScript library that provides the full duplex connection. In addition to providing a consistent cross platform API to access the connection, Socket.io opaquely picks the best available transport mechanism at the time of connection based on the client and server's mutual capabilities. The preferred transports are WebSockets and Flash Sockets, but there are a number of fallbacks including forever iframes and AJAX long polling.

The client also spawns a Web Worker, which carries out the actual computations. Web Workers are a relatively recent HTML5 standard, supported by most modern browsers, that allow a page to spawn real OS-level threads which run JavaScript code. The threads share no state with the page that spawned them, and can only communicate via a primitive message passing interface. Web Workers are ideal for our application because they can perform long running computations without blocking the page. Additionally, their disjunction from the page state is beneficial since the unknown, untrusted user job code that is run within them is effectively sandboxed from the user's browser. We support spawning more than one worker per client, but the default is a single worker, since it's impossible to gather information about the number of cores on a user's machine from within the JavaScript sandbox.

Given the RPC interface and worker, the client logic is quite simple. Once everything is set up, the client fetches a task from the server. The task consists of identifiers for the job and work chunk, the data itself, and the job code, which is either the map or reduce function supplied by the job creator. The client passes the task on to the worker, which evaluates the code and data, capturing emitted key-value pairs. Once the user code completes, the worker passes the accumulated key-value pairs to the client, which in turn sends them back to the server along with the identifiers for the chunk. The client subsequently fetches another task, restarting the process.

### 3.2 Job Server

With easy horizontal scaling in mind, we designed the job server to maintain as little state as possible. The job server maintains only the pool of clients connected to it. Crucially, for each client, the job server keeps track of which work chunk the client has checked out. In our system, client disconnection is the analogue of machine failure. Now.js, via Socket.IO, provides an event indicating a client disconnection. On client disconnection, the job server restores any work chunks that the disconnected client had been working on to the work queue, thus achieving fault tolerance.

The main entry point to the job server is the remote procedure that dequeues a work chunk from a job in the database, associates the chunk with the client, and sends the task to the client. If no chunks are available, the server will wait for a short period of time before retrying. When the client finishes a task, it will call another remote procedure on the server that commits the results to the datastore.

### 3.3 Datastore

We use Redis as our backend datastore for job state. Redis is an in-memory key-value store, and is often referred to as a data structure store because of its native support for atomic operations on sets, lists, and hashmaps, among others. Redis is a natural fit for our application, since our core operations are atomic operations on work queues.

When a user uploads a job, several data structures are created in Redis to support its execution:

- Input Data - A hashmap that maps chunk identifiers to the key-value pair that constitutes the chunk. In the map phase, chunk identifiers are UUIDs. In the reduce phase, the keys themselves are the chunk identifiers, taking advantage of their uniqueness.
- Original Input Data - A raw collection of the input as uploaded by the user, kept in case the job needs to be restarted.
- Work Queue - A list of chunks that still need to be processed, by chunk identifier.
- Intermediate Output - A collection of lists, keyed by chunk identifier, that maintain the replicate outputs for each chunk.
- Final Output - The final list of output key-value pairs, ready to be downloaded by the job creator.
- In Count - The number of unique chunks at the outset of a phase.
- Out Count - The number of fully processed unique chunks. Used in conjunction with the In Count to detect phase completion.
- Runnable - A list of all jobs that have yet to be completed.
- Phase, Map Code, Reduce Code - The current phase of a given job (map or reduce) and the javascript code for that job's map and reduce functions.

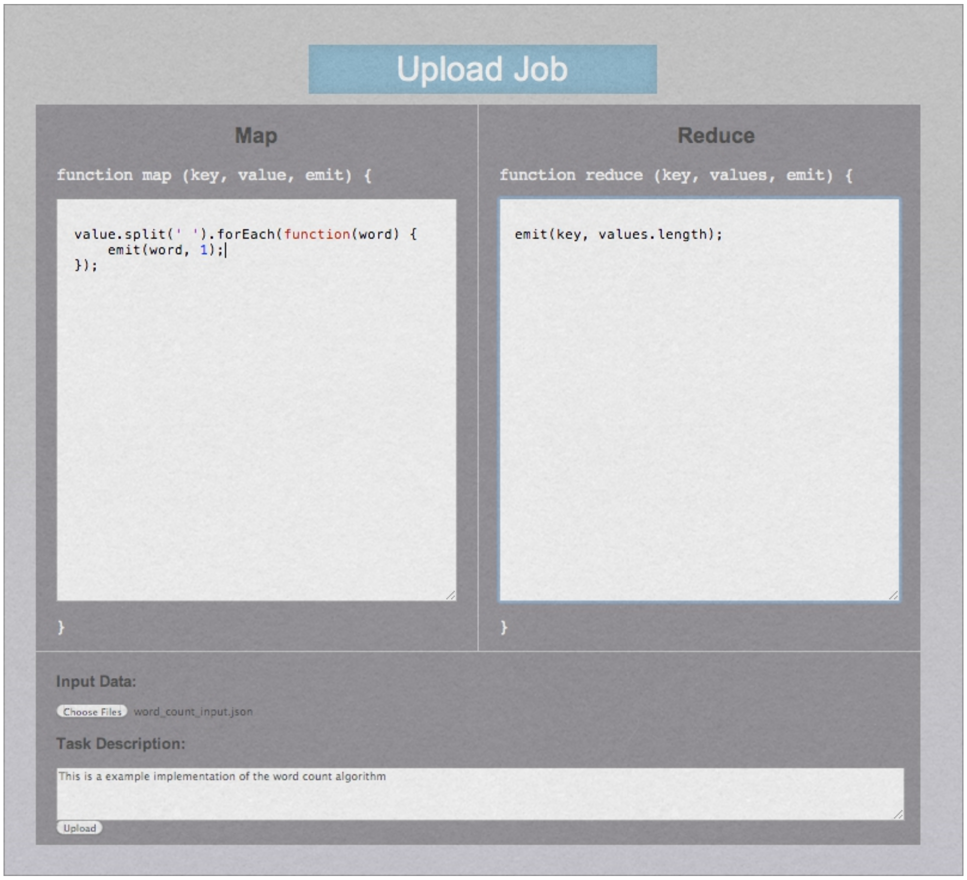
When the client requests a chunk of work to perform, Arke randomly picks a runnable job, and atomically dequeues a chunk UUID from its work queue. It then looks up the corresponding chunk of JSON data in the input data hashmap. This structure allows us to add replication without duplicating the user-provided JSON data. Arke can then look up and ship back the phase and corresponding code to the client.

When the client responds with map or reduce output, the response is enqueued in the list associated with the chunk UUID. If that list contains all replicates, we verify the responses are all equivalent. If so, the out count is incremented. Otherwise copies of the chunk UUID are re-added to the work queue for re-processing.

When the out count matches in the in count, Arke marks the phase as completed, and then completes a phase-specific transition operation. At the end of the map stage, a group-by operation is conducted (the shuffle phase of MapReduce), updating the input data hashmap, work queue, in count and out count accordingly. At the end of the reduce stage, the intermediate data is copied to the final output list, and all intermediate data is destroyed.

### 3.4 Frontend Web Application

Arke would not be complete without a frontend web application allowing users to specify jobs to upload, monitor progress on those jobs and get their results. We currently have a simple web application that allows users to create accounts and perform each of these steps. A screenshot of the upload job page is shown in Figure 1. The input JSON data specified by the user is an array of key-value pairs. Each entry of the array is a dictionary with a key field "k" and a value field "v". Inside of either of these fields, the user is able to specify any JSON-serializable objects.



```
function map (key, value, emit) {
  value.split(' ').forEach(function(word) {
      emit(word, 1);
  });
}
```

```
function reduce (key, values, emit) {
  emit(key, values.length);
}
```

Input Data:

word_count_input.json

Task Description:

This is a example implementation of the word count algorithm

**Figure 1:** A screenshot of the "Upload Job" page of the frontend web application, with the map and reduce code for our word count job specified.

## 4 Testing

This section outlines our testing process. First we describe the two sample MapReduce jobs we used for our evaluation. We chose two very different jobs to illustrate the difference in performance between bandwidth-heavy and computation-heavy jobs. Second we describe our testing infrastructure, including the hardware and software with which both our server and our worker nodes were run. Third we describe how the data was collected.

## 4.1 Sample Jobs

Though Arke's architecture is fundamentally different from that of MRJS, the aims of the two frameworks are similar. Therefore, we are evaluating Arke's performance with the same benchmarks in the MRJS paper. Note that we disabled replication for the purpose of evaluation. Total job time would degrade linearly with respect to increases in the replication factor if it were enabled.

### 4.1.1 Word Count

Word count is a canonical example of a MapReduce job. The Arke user code for word count is provided in Figure 1. Word count has a minimal amount of computation per byte of data, and thus is not well suited for Arke. However, we use it as a benchmark for two reasons. First, as mentioned above, we want to compare directly with MRJS. Second, it allows us to stress our job server and inspect its scalability with relative ease, as clients will hit the job server with a high degree of concurrency.

Mirroring the experimental setup of the MJRS paper, we broke the text of Shakespeare's Hamlet into 100 chunks, where each chunk contains 320 words on average. After the shuffle, there are 8653 intermediate key-value pairs - one for each unique word in the text. Thus, there are 8653 chunks during the reduce phase of the procedure.

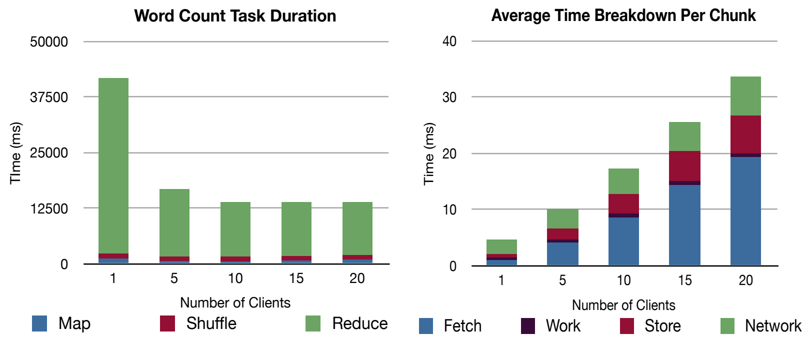


**Figure 2:** Left: The total job duration for the word count job as a function of the number of connected clients. We notice an initial speedup, which flattens out as the job server becomes overloaded with fetch and store requests. Right: The average time spent, per chunk unit, for each phase of operation as the number of clients grows. The work phase denotes the time the client spends executing map and reduce code. The fetch phase denotes the time required to access a task from the Redis datastore. The store phase denotes the time required to store a result in the Redis datastore. The network phase denotes the time between when a fetch phase finished and a store phase began, minus the time the client spent working. We notice linear growth in the time required for fetch and store operations as the number of clients grows.

### 4.1.2 Traveling Salesman Problem

Like the MRJS authors, we implemented a brute force traveling salesman problem (TSP) solver. The solver is exhaustive and considers all possible tours of the input cities, outputting the least costly tour as the final result. Each map task computes the cost of all tours for a subset of all permutations of the input cities. The map outputs the least costly route. The reduce task finds the minimum cost route among map outputs.

Unlike the word count problem, the TSP implementation has very high computation per byte. We mirrored the experimental setup of the MJRS paper, creating a TSP job with 13 cities. This means the mappers must consider 479,001,600 possible tours. Each map chunk considers 5,000,000 paths, giving a total of 96 map tasks.

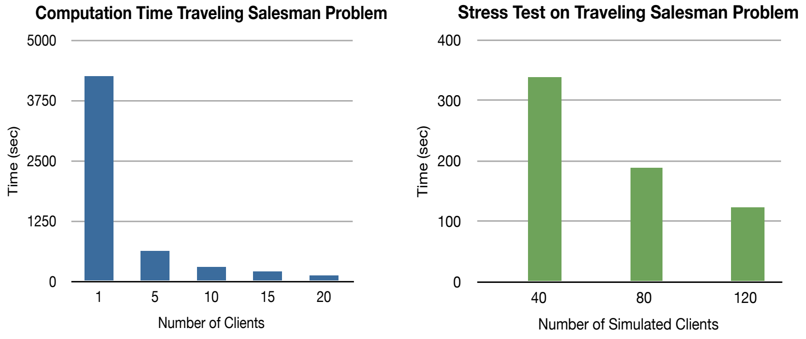


**Figure 3:** The total time required to complete the traveling salesman problem, as a function of connected clients. For high scalability testing, we utilized larger EC2 instances and opened multiple Chrome windows per Chrome instance. The data is separated between the two graphs to account for differences in performance characteristics of the test machines.

## 4.2 Testing Infrastructure

To benchmark Arke, we utilized Amazon's Elastic Compute Cloud (EC2). For both client and server instances, we provisioned the instances with an Ubuntu 11.10 Oneiric EBS boot, using a 64-bit server configuration. The Amazon Machine Image (AMI) was provided by http://alestic.com. To manage deployments of EC2 instances, we utilized boto (a Python interface to Amazon Web Services) and Fabric (a Python library for streamlining the use of SSH for issuing remote commands).

### 4.2.1 Server

The test server was a single EC2 extra large instance. The extra large instance had 15 GB of memory and 8 EC2 compute units. According to Amazon, each EC2 compute unit provides the equivalent CPU capacity of a 1.0 GHz 2007 Xeon processor. Both the Node.js webserver and the Redis datastore were located on the same instance. Should we decide to add additional server instances in the future, the deployment would be straightforward. The server instance is bootstrapped with a simple Fabric script, which could easily be used to add additional instances. The only configuration needed would be to add a load balancer in front of the server instances and offload the Redis database to a shared server accessible by each server instance. Under this scaling model, we would likely maintain an extra large instance for the datastore, but the web server instances could be likely reduced to smaller instances, since the memory footprint of Node.js is small, even with a large number of concurrent connections.

### 4.2.2 Clients

Each test client was an EC2 micro instance. Each micro instance had 613 MB of memory and up to two EC2 compute units.

The browser used on the client instance was Google Chrome (google-chrome-stable 7.0.517.44-r64615). We installed Xvfb (X Virtual Frame Buffer) on each instance to serve as a virtual X server. Using Xvfb, we were able to run Chrome headless (that is, without a display) by specifying the Xvfb instance as the display to use. This allowed us to utilize a browser with a full javascript context, with full support for webworkers, while running without a display.

After requesting the client EC2 instance from Amazon, we run a bootstrap script to install and configure Xvfb and Chrome. The bootstrap script also prints out the public IP address of the newly created instance. To run the tests, we use a Fabric script that concurrently opens an SSH session to each client instance, starts Chrome and loads the URL for our homepage. The client establishes a websocket connection to the server, and begins fetching tasks to perform. This deployment technique allowed us to test with different numbers of clients by simply changing the array of IP addresses used by the Fabric script. However, it should be noted the total runtime of a job as measured by our benchmarking system suffers from including the latency of establishing these SSH sessions and launching the Chrome sessions.

#### 4.2.3 High Scalability Clients

Due to Amazon restrictions, we could only rent 20 EC2 instances at one time. In order to test with 40, 80, and 120 connected clients, we decided to open multiple Chrome windows per instance. To simulate a real connected client, each Chrome window should have its own processor. To accomodate this testing scenario, we needed to upgrade our testing client from micro instances to high-CPU extra large instances. These instances have 7 GB of memory and 8 virtual cores with 2.5 EC2 compute units each. With 8 virtual cores, we were able to open 8 Chrome windows per instance. We tested with 5, 10 and 15 instances, to simulate 40, 80 and 120 simultaneous connections. Since the computation power of these instances is dramatically different than the computation power of the micro instances, we gathered this data in a separate collection.

We also modified the TSP test to accomodate a larger number of clients. With only 96 work chunks, testing with 120 clients would have wasted worker nodes in the system. We modified the TSP problem to contain 14 cities. This means the mappers must consider 6,227,020,800 possible tours. We maintained the map chunk size of 5,000,000 paths, giving a total of 1246 map tasks.

### 4.3 Data Collection

To record metrics on Arke's performance, we used a timestamp-based logging system. We recorded a timestamp at each of the following event boundaries:

- Start fetch: Server receives request, begins Redis fetch for task.
- End fetch: Server fetched task from Redis, begins send to client.
- Client start: Client receives task from server.
- Client end: Client finishes task, begins send to server.
- Start store: Server receives result, begins Redis store.
- End store: Server finishes store.

Whenever we calculate deltas between two events, we must be certain the two events were logged on the same machine. This is because clocks across machines are not guarenteed to be synchronzied.

We ran the tests in an environment in which only one job was in the datastore at a time. When the job was marked as finished, the server would send a message to connected clients, asking them to send their logs back to the server. The server would then aggregate all logs across all clients, including the logs kept on the server, and write the output to file. We utilized a Python script to parse the contents of the log file and generate the results presented in section 5.

## 5 Results

Our results demonstrate Arke's ability to harness the power of increasing numbers of worker nodes for both the word count and traveling salesman problems. When running the word count job, there is a substantial speedup of roughly 300% when switching from 1 to 5 worker nodes. We show in Figure 2 that while the

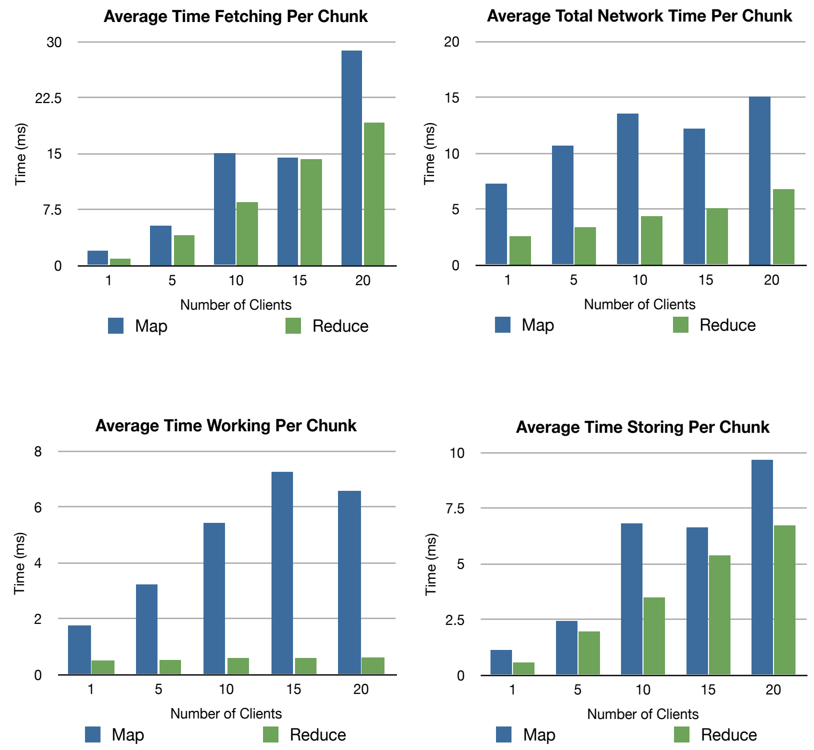


**Figure 4:** The average time spent on each phase of operation for both the map and reduce phases of the word count job.

amount of time spent working per chunk stays roughly constant, server-side fetch and store operation performance degrades linearly. We believe that this degradation is due to a linear increase in the average length of the Node.js event loop. This causes slower fetch and store times, as shown in Figure 3. We hypothesize that adding additional Node instances behind a load balancer would reduce the coefficient of linear growth on this graph, and therefore continue the word count speedup even past 5 client nodes. Nonetheless, we believe we have shown that for an I/O intensive job like word count, Arke demonstrates better scalability than MRJS. We believe the variation in time spent working per chunk is due to variation in EC2 micro instance performance.

The results shown in Figure 4 show Arke's exemplary performance under the conditions for which it was designed. The brute force algorithm we used to compute results for the traveling salesman problem has significantly higher computation per byte than word count. As a result, there is less I/O strain on the Arke job server, eliminating bottlenecks and providing perfect scalability. Our stress testing shows that Arke can perform beyond the bounds of MJRS as Arke continues to scale very well with a near linear speedup for up to 120 simulated worker nodes. Furthermore, we utilized only a single EC2 instance for our web server and datastore for these tests. We believe Arke could scale horizontally to accomodate thousands of simultaneous connections.

## 6 Future Work

Arke is still in the proof-of-concept stage, and as such, there is a large array of potential improvements and directions for exploration. We begin with a discussion of near-term improvements to Arke's core computational framework, and then move on to considerations related to deploying Arke as a public service.

## 6.1 Core Framework

### 6.1.1 Prefetching Tasks

Currently, Arke is set up to distribute one task to each client at a time. The client requests a task, the server fetches and serves the task, the client computes the results and sends them back to the server, and the server stores the results. These operations happen in order over and over again, with no overlap. This model wastes client resources, because network and CPU are not co-utilized. Note that WebWorkers, though they exist in their own thread, are still a conventional JavaScript environment — single threaded and evented. As a result, the worker's computational resources would be consumed optimally by passing a work chunk to the worker precisely when the previous chunk completes. By keeping a running average of the network latency and compute time for job it has worked on, the client could calculate and update a time delta at which to fetch a new task, ensuring that new tasks arrive at approximately the same time a previous task completes.

### 6.1.2 Chunking Phase Input Data

We chose not to chunk map and reduce input data in the sense that our work chunks consist of a single key-value pair, although users may manually chunk the map data before they upload it. Chunking data cuts down the maximum number of workers that can concurrently work on the same job, but for some jobs larger work chunks may lead to better performance. We would like to experiment with heuristics for automatically chunking intermediate key-value pairs, or alternatively make chunking parameters a user-configurable option.

### 6.1.3 Combiners

We have yet to implement combiners, a part of the MapReduce specification that allows for preliminary aggregation of intermediate key-value pairs. A user would upload a combiner function along with the map and reduce functions, and the combiner would be run at the conclusion of computation on a map chunk. Combiners would reduce stress on the shuffle phase of our job flow.

### 6.1.4 Straggler Handling

Stragglers are nodes that have poor performance and increase the overall runtime of a job by taking a long time to return their results. In our testing setup, worker nodes were equivalently competent, as they were all the same level of instance on EC2. Therefore, we didn't run into any problems with stragglers. However, the computational resources of computers across the open internet varies widely. Therefore, we believe it is essential that Arke handle slow nodes. To handle stragglers, we would need to address two challenges: first, to develop heuristics for identifying straggler nodes, and second, to modify our job control structure to allow redistribution of straggling tasks to more competent nodes.

### 6.1.5 Stateless Job Servers

In our current implementation, a job server crash is fairly disastrous, because we lose track of which chunks were checked out to clients connected to that server. We would like to make the servers entirely stateless by storing checked out chunks in lists in the datastore. The job servers could send heartbeats to each other, and restore lost chunks in the case of failure.

### 6.1.6 Horizontal Scalability

We designed Arke with horizontal scalability in mind, but we have yet to test scaling out the job server and datastore. As mentioned in the evaluation section, we believe that data gathered from the word count task indicates that spinning up additional job servers would result in linear scaling with regard to total time per task as each server's event loop becomes less and less crowded. We would need to set up the Node instances behind a load balancer and ensure that clients could establish a websocket connection directly to the Node instance the load balancer assigned it.

## 6.2 Arke as a Service

### 6.2.1 Security

In order to deploy Arke as a full-featured web service, we would need to perform data validation for uploaded jobs. Currently, Arke trusts the user-provided JSON data and map and reduce functions as units of work shippable to volunteer clients. Erroneous JSON data or code could cause a job to remain in the datastore indefinitely, since clients could never make progress by successfully computing results. In addition, a thorough security review should be conducted.

### 6.2.2 Backbone

If we were to deploy Arke as service, we would want to guarantee to potential users that there is a persistent backbone of workers to process their jobs even if volunteer nodes across the open internet are lacking.

### 6.2.3 Embeddable Application

We believe that making Arke embeddable into third-party websites would be a significant boon to its practicality. Other websites could donate their users' resources by embedding Arke within their site's pages. A small widget could indicate Arke's status and allow the user to disable it.

### 6.2.4 Generic Task Platform

By exposing an open API for other developers to fetch tasks and store results, we could enable the creation of arbitrary Arke clients, eliminating the restriction that jobs can only be expressed as map and reduce code executing in the context of a Web Worker. For example, this would enable human-directed tasks to be run atop the Arke backend.

### 6.2.5 Social Incentivization

Various applications connected to social networks like Facebook have proven that users will put in significant time and resources for virtual rewards. Therefore, we think we could increase user adoption of Arke by making it visible via social networks and granting rewards for volunteering as a worker.

# 7 Conclusion

In this paper we have presented Arke, a MapReduce-based framework for distributed computing on a dynamic cluster of volunteer nodes across the open internet. Our results demonstrate that our evented approach to serving jobs has more potential for scalability than MRJS, a prior attempt at such a framework. Furthermore, we have integrated fault tolerance, replication, and support for running multiple jobs simultaneously.

As discussed in the future work section, we believe there any many possibilities for improving and deploying Arke; that said, the current implementation shows great potential. Arke targets a domain of problems distinct from typical MapReduce jobs — it would only be practical for problems with a high amount of computation per byte, and a large number of input chunks. However, with the integration of the our ideas in the future work section, we think Arke could harness a large amount of computational resources that are idly wasted. Arke could therefore provide a free and open platform for computation on clusters of previously unachievable size and scope.