Midterm Report

Distributed Processing System with MapReduce Model on Browsers and Node.js

101-2 Special Topics on Cloud Computing, CSIE, NTU

Tzu-Hsien Gao, Tai-Lun Tseng

Email: {r01944033, r01922094}@csie.ntu.edu.tw

0.1 Preface

This report summarizes the collective results on the project so far. It includes the original project description, survey results on other projects, roadmaps for implementation, and some possible issues for the project.

While the report is aimed to become a milestone of works done in the midterm, most parts of this report will reappear in the final project report.

Section 2 and 3 gives motivation and description of the initial idea. Section 4 summarizes the survey result of several open source projects and web-related techniques that are useful in the project. Section 5 and 6 describe the design of the system architecture, ideas of implementation and possible upcoming issues during implementation.

0.2 Motivation

MapReduce[1] provides a distributed way for processing large-scale data. In traditional MapReduce implementations, a server node dispatches data and tasks to various client nodes and collect results after mapper and reducer functions are done by clients. Both server and client need efforts to configure connection and execution environment for MapReduce programs.

In order to get many clients in a more convenient way, we introduce a new kind of MapReduce client, i.e. browsers. In the initial scheme, researchers who need more computing resources do not necessary set up lots of clients; instead, they can use a HTTP server that establishes connection with clients visit the

researchers website (server). Data, mappers and reducers can therefore dispatch to those browsers and execute. To support a research, a resource provider only need to keep a browser opened and connected to the server. And in the future, we can extend the functionality more by enabling other node.js[2] applications to become clients; thus researchers who are doubt for the speed of browser computing can set up client machines with node.js and libraries we provided.

0.3 Project Description

This library runs a program using MapReduce programming model on the server and dispatch mapper/reducer tasks to many registered client browsers program.

To register, clients visit a site hosted by the server and accept worker permissions. Javascripts of the site will create several HTML5 workers. They will be mappers or reducers (a browser client may has mapper and reducer workers concurrently), waiting for dispatching tasks and data from the server. Upon receiving works, the workers become active, execute the map/reduce works, return results back after finsihing execution repeatly.

Different with the traditional MapReduce model, we cant communicate between browsers since the standard of communication between browsers is not popular in this time, so the the intermediate result done by map workers will be report to theselves, and then shuffle these intermediate results to the reduce workers.

Also there will be an API for node.js that provides adapters connecting server-side javascripts and node.js computing environment. Eventually the server can also dispatch tasks to node.js servers, i.e. treating node.js as a powerful MapReduce computing nodes.

0.4 Surveys on Related and Supported Works

In order to achieve our motivations, it is necessary to give a brief survey first to check how other projects implement the similar goal, and compare differences with our approach. The survey result can be categorized into two parts: *MapReduce-like implementation* and *javascript libraries and techniques*. For the first part, we try to look especially into MapReduce-like projects that is implemented in javascript.

1. Atomize.js[3]: It is amied to provide distributed objects and atomic transaction feature between the server and all clients. The clients register to server and read/write variables with server; Atomize.js ensures that the variable is treated as "global" and identical among all clients. The descriptive sample given by authors of Atomize.js is a simple online game that uses atomic feature to manage game states. For client-server connection it uses SockJS library introduced below.

2. MapRejuice[4]: MapRejuice is a simple implementation on MapReduce algorithm. It builds a web server that delievers javascript contents and use RESTful API as entrances for receiving outputs of map and reduce functions. It uses Socket.io for client-server connection.

3. WebWorker[5]: WebWorker, a.k.a. Web Worker API, is the well-known feature that comes to new browsers with HTML5 standards. Specified by W3C, the objective of WebWorker is to provide concurrent programming in browsers; therefore heavy-loaded scripts can be executed in the worker thread and it does not block the main thread of browser, which leads to no-response feels to users. In our approach, it is natural to execute map and reduce functions on a worker thread in order to speed up performance in the browser.

4. WebSocket[6]: Similar to WebWorker, it is included in the HTML5 standard. As the name shows, WebSocket provides socket programming functionalities to browsers, which traditionally rely on HTTP protocol communication. WebSocket is more suitable for modern web use cases because while HTTP protocol is stateless and disconnects after response sent/received, WebSocket can create a tunnel between the server and browsers to exchange information rapidly and simutaneously. Again it is natural for our project to use WebSocket as data exchange.

5. Socket.io[7]: Although WebSocket provides socket-connection abilities to browsers, it currently has only plain API and unable to support old browsers. Socket.io tries to solve these problems with easy-to-use interfaces that wrap WebSocket APIs and use Ajax instead of WebSocket when older browser is detected.

6. SockJS[8]: It is a wrapper to WebSocket. When SockJS detects that the browser is too old to provide WebSocket functionality, it uses other methods (e.g. Ajax) to emulate socket instead. This is very similar to Socket.io, which is more popular. They both have browser/server side implementation.

0.5 System Architecture

In this section we describe the overall architecture that covers main components of the system, as well as go trhough the execution overview and discuss those components in detail.

0.5.1 System Overview

The system is combined with serveral parts:

(I) the master server

(II) the client browsers

For (I) we use node.js as its basic infrastructure.

For (II), though Socket.io and SockJS can give a unified interface among WebSocket enabled/disabled browsers, only WebSocket enabled browsers are allowed to use the system. Thus, the connection for MapReduce algorithm flow between (I) and (II) is done via HTML5 WebSocket API.

0.5.2 Execution Overview

While designing the MapReduce model for the server-browser architecuture, we compromise for some additional steps (*configuration*, *signal* and *collect*) and efforts (in *split*, *shuffle* and *reduce*) to the original model due to some constraints of our architecture.

1. *Configuration.* User sets up some properties of the execution flow, e.g. data split strategy (see below).

2. *Split.* Master server split data into multiple chunks. By how configuration varies the system either splits data as fixed size, or dynamically determines number of clients and splits data to the same number. The data is split into M chunks.

3. *Dispatch.* Master server dispatch M chunks to M clients, as well as the map functions and reduce functions.

4. *Map.* The map functions take individual chunk as input, and output a set of key-value data corresponding to the chunk. Since we are using javascript the key-value data can be easy represented as a javascript object or a JSON.

5. *Shuffle.* Clients return the map results via WebSocket, and master server immediately send it to different clients by sending different key-value data to client reducer that corresponds to the key.

6. *Reduce.* Upon receiving key-value data, clients run reduce function to summing up values of the key. The data does not come at the same time so the reducer will wait for a signal referring to the end of execution.

7. *Signal.* When master receives and re-dispatches all data from every map functions, it sends END signal to all reducer clients, telling them there is no further inputs and thus reducers can terminate the process.

8. *Collect.* After reduce function is terminated, the client sends the final key-value result back to master server, which collects them and writes to the final output.

0.5.3 Part I: Master Server

The master server does two major works: (1) provide web content delievered to browsers, and (2) control the whole MapReduce algorithm process. We discuss them separately.

1. Web Content Delievering
   The node.js server creates a HTTP server object and listens for HTTP GET requests. Upon recieving requests, the HTTP server object sends web content back via HTTP response.

   The web content contains the following parts:

   - HTML content: a webpage that allows user to register to server.

- Browser-side javascript program: the main component that executes MapReduce related works.
- Other assets files.

Details of the HTML content and browser-side javascript program will be discussed in Part II.

2. MapReduce Algorithm Process Control
   The server begins the algorithm by firstly dispatches split data, map function and reduce function to registered clients.

   In theory after all data are sent, the dispatch process is done and the server starts the shuffle process; but since node.js has a non-blocking I/O characteristic, the server runs the shuffle process in a callback which is called when map function returns the results.

   For the shuffle process, the server first marks the returning client as *MAPEND* state; second, for each key-value data from mapper, it uses bucketing algorithms (described below) to decide which reducer should the key go. Finally it dispatches the key-value data to that reducer.

   Upon all the clients are marked as *MAPEND*, the server knows that all initial data are mapped to key-value data, so it sends *END* signal to every reducer.

   Finally, a callback awaits: when the reducer returns key-value data back, the server calls the collect process that writes the data to user-defined output location e.g. database or static files.

0.5.4 Part II: Browser Clients

After recieving web content from the server (or, with another word, the user visits the website hosted on the server) the user can register to the server to wait for coming tasks.

The register action builds WebSocket connection between browser and server, and the server will add the browser to one of its vailable client; the action will also create a WebWorker object (a.k.a. worker thread for below) that is initially pending.

When the browser recieves the split data, map function and reduce function, it starts to execute mapper function on the worker thread, split data as the input.

After the mapper terminates it produces an array containing many key-value objects. The worker thread uploads the array via WebSocket.

When the upload process is done, it starts to execute the reduce function by wrapping a callback over it: only when a key-value data is sent from server will the callback be called and the reducer be executed, key-value data as the input.

When the browser recieves the *END* signal instead of data from the server, the worker thread should terminates the callback wrapper and starts to upload the final result.

0.6 Issues

This section has two parts: first we give some open issues that are not coming up to solutions until now, and second we discuss refinement methods that can be taken to improve performance, stability and usability of the system; some of them might solve the open issues.

0.6.1 Open Issues

One of the severe problem in distributed computing is the Internet latency. While distributed computing aims to speed up computation by using multiple machines as computing resources, the communicaiton would cost a lot if the problem is not dealed carefully.

For the original MapReduce model, the computation takes place on a cluster of computers that are connected closely[1]. But our system uses web browser as computation sources, thus performance would be severely reduced by latencies between server and browsers.

The second possible bottleneck is the execution speed of javascript codes. Javascript is usually executed in a built environment that can be abstractly described as JSVM; famous examples include Google V8 engine[9], Mozilla Rhino[10], etc. Javascript runs dynamically with an interpreter and does not perform well comparing to compiled programming langauges; moreover, most of the javascript engines in browsers have only limited compute ability and might slow down the overall performance.

Another issue is the performance evaluatation and demonstration part: it is difficult to find a suitable benchmark or test cases that can fairly evaluate our system. Finding lots of browsers to do a demonstration is another hard task.

0.6.2 Refinements

- Key-to-Reducer Bucketing
  The key-value data produced by the mapper should be delievered correctly to the reducer corresponding to the key so that there will be no data with identical key that is delievered to different reducer. To ensure the consistency a bucketing algorithm is used on master server while re-dispatching data to reducers.

  The bucketing strategy must first ensure the consistency, and then try to amortize the number of responsible keys for each reducer, due to the load balancing objective.

- Fault Tolerance
  It is quite common that clients are disconnected from the server in our design. Although we choose WebSocket as the way of task dispatching, it resides under the webpage within the browser - or, more specifically - a tab of the browser; once the tab is closed by the user or other reasons like browser crash (which is sometimes unevitable since some browsers e.g. Safari clears contents of every tabs if it faces memory-insufficient states), the WebSocket and Web Workers fail altogether. The following works will fulfill the weakiness.

- Mapper Streaming
  In the implementation above the mapper sends output data after the whole map function process terminates. But this produces two problems: first, the output data is relatively big for master server to do a transfer task between mappers and reducers; second, this will decrease performance since the scenario implies that master server must wait for a mapper terminates in order to retrieve its output.

  To leverage this disadvantage, we introduce a method trying to confront it: for a mapper in process, it periodically sends its current output to master server instead of sending all of them after whole process terminates. The idea is different, while similar in concept, with the streaming technique.

  To be more detailed, we will predefine a chunk size; when the mapper produces data that reaches the chunk size we use another thread to send the data to master server. The chunk size chosen is an important factor in our approach: too large chunk size distracts the goal that we want to confront the cons, while too small chunk size increases lots of additional connections, which is unefficient.

  For our implementation, we will first adopt the initial approach that sends data after the mapper terminates; and then we will use the streaming idea for refinement. There will also a comparison on performance within first approach and second approach with different chosen chunk sizes.

Bibliography

[1] Dean, Jeffrey & Ghemawat, Sanjay *MapReduce: Simplified Data Processing on Large Clusters*, 2004

[2] `http://nodejs.org`

[3] `http://atomizejs.github.io`

[4] `https://github.com/ryanmcgrath/maprejuice`

[5] `http://www.w3.org/TR/workers/`

[6] `http://www.w3.org/TR/websockets/`

[7] `http://socket.io`

[8] `http://sockjs.org`

[9] `http://code.google.com/p/v8/`

[10] `https://developer.mozilla.org/en/Rhino_Shell`