Final 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

1 Abstract

This report summarizes the collective results of the project. It includes the original motivation, project description, survey results, system design and implementations.

Section 2 describes the motivation of the initial idea. Section 3 gives a brief description of the project. Section 4 summarizes the survey result of several open source projects and web-related technologies that are adopted in the project.

Section 5 and 6 describe the system architecture design and some details in the implementation. Section 7 discusses some works that may be done in the future, including some unsolved problems due to technical limitations as well as some possible directions that can extend the project scope.

Section 8 lists related works of the project; finally, we put some conclusion at section 9.

2 Motivation

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

In order to get many clients in a more convenient way, we introduce a new kind of client, i.e. browsers. In the initial scheme of our idea, researchers who need more computing resources do not need to set up lots of clients; instead, they can use a HTTP server that establishes connection with clients who visit the website (server). Data, mappers and reducers can therefore dispatch to those browsers and execute. To support a research, a resource provider only needs to keep a browser open and connected to the server.

Due to the lack of cross-browser communication mechanism in the current HTML standard, the intermediate data have to send back to the server and then dispatch to clients again. This will become a bottleneck.

And because the server and browsers communicate via HTTP protocol, the interaction between the server and clients is much slower than in clusters. Also the browser is not a fast execution environment. These challenges should be considerd during implementation.

We can extend the functionality by enabling other Node.js 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.

3 Project Description

The project implements a basic structure of the MapReduce programming model. The master server is written in Node.js with Express framework, which helps faster developing on Node.js; the webpage delievered by the server is treated as the client nodes. The core of the mapReduce algorithm, altogether with utilities like input reader and output writer, are written as a Node.js module that is used by Express.

To register as a computing node, clients visit the site hosted by the server and a socket connection will be established. The client then becomes a node a waits for incoming requests.

To start the MapReduce program, administrators visit another master page on the server and click the Start button on the page. The server will then start to read input, dispatch tasks and wait for result and write to output directory.

Different with the traditional MapReduce model, we cant directly communicate between browsers, so the the intermediate result done by map workers will be sent to the server fist, and then shuffle these intermediate results to the reducers.

4 Technologies and Open Source Projects

After several surveys, we implement the system with some open source projects from the survey result. Here we describe the technologies and open source projects that are used in our project:

1. Node.js: Node.js is a server-side software system that aims to be a basis of scalable Internet applications; built on V8[10] javascript engine developed by Google, it has an event-driven and non-blocking I/O nature. Since it is written in javascript, we use it to develop server-side program in order to provide a homogeneous environment between server and clients.

2. WebSocket[7]: The WebSocket technology is included in the HTML5 standard. It provides interfaces that enable socket programming functionality

to browsers. WebSocket is more suitable for modern web use cases; 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. For our project, it is natural to use the mechanism for data exchange. But since WebSocket does not support older browsers and it has unfriendly APIs, we use Socket.io, an open source library that solves these problems.

3. Socket.io[8]: 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.

   The interface of Socket.io is purely event-driven and message-passing. To interact with each other, the sender emits a message with a specified name and the message body; and the receiver must register an event listener with the name to catch the message.

   Moreover, Socket.io gives programmers several additional useful abilities. The most important ability that benefits our project a lot is the callback mechanism: when the receiver gets the message, it can notify the sender that the message has been received via a callback function. For convenience, we call the mechanism as *handshaking* in the following sections.

4. Underscore.js[11]: Underscore.js is a javascript utility library that provides various features not included in the original language, e.g. object iterator, splitter, comparison, etc. Some parts of the program are benefit from the help of it.

5. Express[12]: Express is a web application framework that makes Node.js more easy to provide webpage contents as well as server-side routing and flow control. We use Express to build the two webpages and trigger the MapReduce program.

5 System Architecture

In this section we first describe the overall architecture that covers main components of the system, and then gives an detailed execution flow of the MapReduce algorithm. After the execution flow, we describe both the master server as well as the clients in detail, and the API specification between them.

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) the browsers first retrieve contents from the server, and then use the javascript in the content to establish the connection via Socket.io; thus it can wait for incoming tasks.

5.2 Execution Flow

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.* The server records the clients currently registered and lock the incoming registeration during the MapReduce program executes. Clients are put into a client pool in convenience of being used by the following steps. Here we let M be the number of clients in the pool.

2. *Split.* Master server reads and splits the input data. By default, the data is not split by the server; instead it is user's responsibility to split data into multiple files and put them into the input directory. Here we let F be the number of input files.

3. *Dispatch.* Master server dispatch F files to M clients, as well as the map functions and reduce functions. To armotize the number of files received by each client, the ratio r = ceil(F / M) is applied to each client so that they all receive r files in average.

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 Socket.io, and master server immediately send each key-value pair data to different clients. For each intermediate key we assign a reducer that is responsible to reduce the key; in order to do this, we use a mechanism similar to bucketing, and will be described in the next section.

6. *Reduce.* Upon receiving key-value data, clients run the reduce function to sum 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 the execution.

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

8. *Collect.* After the client sends the final key-value result back to master server, the data will be written to the output directory. A brief summary will be shown indicating execution time, number of keys, number of clients, etc.

5.3 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 for root path. 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 the next section.

   There is another webpage on /master path that plays as a role of MapReduce algorithm trigger.

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

   Due to the non-blocking I/O and event-driven characteristic of Node.js, the following processes are not execute after the data are dispatched; instead, we let the server listens to several messages that each represents a part of the algorithm, such as *MAPDATA* (in order to receive mapper output), *REDUCEDATA*, etc. The messages are treated as APIs and will be discussed later.

5.4 Browser Clients

After recieving web content from the server (or, with another word, the user visits the website hosted on the server) the client is registered to the server to wait for coming tasks; the register action builds Socket.io connection betweent the browser and the server.

When the browser recieves the split data, map function and reduce function, it starts to execute mapper function where split data is the input. Upon receiving, the mapper and reducer functions are just plain strings, but javascript provide convenient ways to translate them back to executable functions.

After the mapper terminates it produces an javascript object containing many key-value pairs and upload those intermediate data to the server.

The reduce function is executed while the client receives the *REDUCE* message. The message will hold key-value pairs as message body. The reduced output is stored by the client.

When the browser recieves the *MAP_ALL_END* signal, it then starts to upload the final result after all reducers are executed.

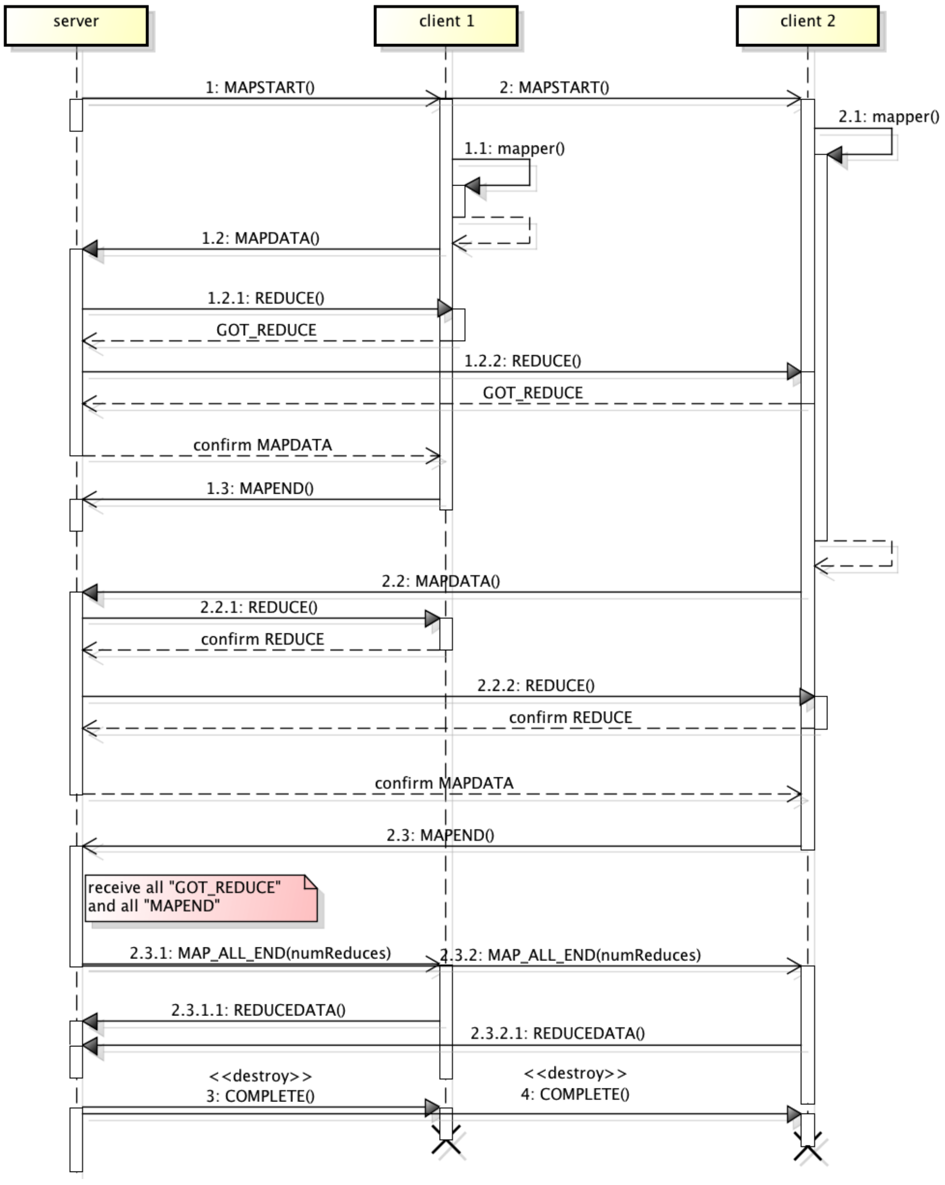
5.5 API Specification

Socket.io uses message-passing mechanism to communicate and send/receive data. Different types of messages are identified via its message name; therefore we build a set of message types that is used as APIs that help the server and clients interact between each other.

Here shows the complete list of APIs:

API name	Direction	Description
MAPSTART	Server to client	The server sends split data, mapper, emitter and reducer functions to the client and tells the client to start mapper task. Note that the mapper, emitter and reducer in the message body are just plain strings, but can be translated back to functions. The emitter will be executed inside the mapper function in order to summarize the outputs by the mapper.
MAPDATA	Client to Server	The client sends the mapper output to the server. Upon receiving MAPDATA, the server will use handshaking to notify the client, which will then emits the MAPEND message.
MAPEND	Client to Server	The client tells the server that the mapper phase has completed.
REDUCE	Server to Client	The server sends the message with a key-value pair, and tells the client to execute reducer task by using the key-value pair as the input.
MAP_ALL_END	Server to Client	The server tells all clients that (1) all mapper tasks are done and (2) all REDUCE messages has been sent and *received* by all clients. Thus the clients can therefore upload the output data of reducers. This message will be broadcasted while satisfying two conditions: (1) the server receives all MAPEND messages. (2) the server knows that all REDUCE messages has been received.
REDUCEDATA	Client to Server	The client uploads the output data of reducer to the server. The message will be sent while the client receives MAP_ALL_END message. If there are still some reducers running, the message will be sent right after all reducers are terminated; otherwise it will be sent directly after ceceiving MAP_ALL_END. Upon receiving the message, the server uses handshaking to notify the client, which will then emits the REDUCEEND message.
REDUCEEND	Client to Server	The client tells the server that the reduce phase has completed.
COMPLETE	Server to Client	The server tells all clients that the whole program has completed. A brief summary is sent in the message body.

After explained those APIs in detail, we illustrate the overview of the execution flow, which is complicated due to the communications between the server and the clients. The sequence diagram below describes the execution in detail.



6 Implementation Details

In this section, we provide some implementation details in order to reflect many design and implementation challenges of the project.

6.1 Input Reader at a Glance

Node.js provides File System module that lets programmers read and write on the file system of the server. We use it to read input data. Again, due to the nonblocking I/O nature of Node.js ,there are two types of reading files and directory - synchronous and asynchronous versions.

To use the File System module, simply writes `var fs = require('fs');` to retrieve the module. We use the asynchronous version of reading directory, `fs.readDir()` ; for reading files, however, we use the synchronous version `fs.readFileSync()` which blocks the execution of the server until the file is read.

6.2 Key-to-reducer Bucketing

For our server-browser model, the intermediate output data from mappers are sent to the server first and then dispatches to reducers. For each key generated by the mappers, it should be assigned to a specified reducer so that the reducer can sum up the value of the key; so, it is important that for any key-value pair data, they should be sent to the same reducer if the keys is identical.

To achieve this, we need an assignment, i.e. a mapping, from keys to reducers, in the shuffle step. The assignment must ensures that the same key goes to the same reducer; moreover, the assignment should amortize the number of keys each reducers received in order to achieve good load balancing.

We use an idea similar to the bucketing to solve this issue. A key pool is given and initialized as an empty object in the congiguration step; to retrieve a client with a given key, the pool accepts the key as the input and

(1) checks whether the key is already in the pool;

(2) if the key is not in the pool, it assigns a new client ID (retrieved via an iterator on the client pool) to the key and store the key-client ID pair into the key pool. Finally the pool returns the ID;

(3) if the key is found, the pool returns the client ID assigned on the key.

Note that in (2) we ensures the load balancing by using the iterator on the client pool, since the iterator will go through clients in the pool one by one; for next incoming new key, it will be assigned to the next client.

This is a relatively simple bucketing scheme but works well, for both mapping accuracy and load balancing.

6.3 Data Synchronization Problems

Among the whole program, we use event listener to catch messages; but the mechanism is difficult for flow control especially when there are synchronization problems. For our MapReduce model, there are 3 tricky parts that needs to take care of:

1. If the MAPEND message sent by a client is received by the server before the MAPDATA message sent by the client is received, the data in MAPDATA message might be ignored by the server.

2. If the MAP_ALL_END message is received before any REDUCE messages received by clients, the data in REDUCE message might be ignored by the client.

3. Clients do not know how many REDUCE messages will be received, so we need a mechanism that notifies the client to stop listening to new REDUCE messages and start to upload the result.

In this section we describe solutions that solves each issue.

6.3.1 MAPDATA and MAPEND Ordering

To ensure the MAPDATA message is handled by the server before receiving the MAPEND message, we use the handshaking mechanism provided by Socket.io to enforce the message order:

```
socket.emit('MAPDATA', {
    data: mapperOutput
}, function () {
    socket.emit('MAPEND', {
        keys: _.keys(mapperOutput)
    });
});
```

The code snippet comes from the client side javascript. When the MAPDATA message is received by the server, it first shuffles the `mapperOutput` to reducers, and then calls the callback function, i.e. the second input argument of `socket.emit()`.

This mechanism is also adopted in the case of REDUCEDATA and REDUCEEND messages ordering.

6.3.2 REDUCE and MAP_ALL_END Ordering

As described in the API section, the MAP_ALL_END message should be sent when

(1) the server receives all MAPEND messages;

(2) the server knows that all REDUCE messages has been received.

For (1) the server simply uses a counter to record the number of incoming MAPEND messages and detect whether the number of the counter equals the number of clients.

For (2) we propose a counting scheme that counts before and after the REDUCE message sending. In the configuration step we initialize two objects `send` and `got`, and another object `reduce` that contains the two objects as properties.

Before the server sends the REDUCE message to client x, let `reduce.send[x]` increase by 1 (if `reduce.send[x]` is undefined then set the value to 1). After the client receives the message, it uses the handshaking mechanism to notify the server; the server then increases `reduce.got[x]` by 1.

On another side, when the server receives all MAPEND messages it immediately knows how many REDUCE messages will be sent in total. We call the value N. Therefore, when the server collects all MAPEND messages, `reduce` object starts to observe two events:

a. `reduce.send` is identical to `reduce.got`, i.e. they have identical keys and their corresponding values are all equal to each other;

b. the total values in `reduce.got` equals to N.

Upon the two events occured, the server can send the MAP_ALL_END message. By using these counting techniques, we can ensure that the MAP_ALL_END message is sent after all REDUCE messages are received by clients.

6.3.3 REDUCE Event Listener Problem

Recall the `reduce.got` object above. When the two conditions are matched, `reduce.got[x]` represents the number of REDUCE messages sent to the client x.

So the problem is easy to solve by using `reduce.got` well; we put `reduce.got[x]` in the message boy of the MAP_ALL_END message toward client x. Upon receiving the message, client x knows the number of REDUCE messages that it will got; therefore, it can send the REDUCEDATA message when all reducer tasks are done by simple counting.

7 Future Works

Fault tolerance is the biggest unsolved issue in the project. MapReduce needs a lot of data transmitions. If one connection node failed, the whole algorithm may break. What's worse, it is normal for a client disconnects from the server, so fault tolerance should be carefully considered; but now we are unable to come up a good solution for this issue.

For benchmark and evaluation, our model is hard to evaluate and do performance comparison. Our system needs different benchmarks from traditional MapReduce model, and is unsuitable to compare with traditional models.

Peer-to-peer (P2P) communication is needed to transmit the intermediate data between mappers and reducers. WebRTC[2] will be implemented by most of the browsers in the future, which is a solution to enable browser-to-browser data exchange.

In the original idea the mappers and reducers are excuted via WebWorker[6], which provides multi-threading javascript execution functionality in browsers. If mapper and reducer tasks are heavy, they can be executed on the worker thread, but the data synchronization issue between main thread and worker thread is hard to solve so currently it is not implemented in the project.

8 Related Works

MapReduce[1] proposes the original idea of the MapReduce programming model; our project tries to imitate the execution flow described in the paper.

Atomize.js[4] provides a distributed object and atomic transactions on it. Although not related to MapReduce, it uses SockJS[9], which is similar to Socket.io, to send and receive messages. Some of the API specification of our project is inspired from this project.

MapRejuice[5] is a simple implementation of the MapReduce algorithm on server-browser, which is similar to our design; it uses several RESTful APIs as the entrance of mapper and reducer, e.g. `/map` and `/reduce`. Our project provides more functionality and flexibility, comparing to MapRejuice; also we use Socket.io to exchange data instead of URL redirection in MapRejuice model.

9 Conclusion

This report describes a system running MapReduce model on browsers and Node.js. The system provides a low-cost and high-scalable processing environment using HTTP server and browsers, so clients have no effort on configuration.

We implemented a server-browser MapReduce system to reduce the cost for clients setting comparing to the traditional MapReduce scenario. Researchers can easily process large dataset with a parallel, distributed algorithm in our model.

The final result of our work is open-sourced and hosted on github:
`http://github.com/teaualune/1012NTUSTCC-PRJ`

References

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

[2] `http://www.webrtc.org/`

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

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

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

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

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

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

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

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

[11] `http://underscorejs.org`

[12] `http://expressjs.com`