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 on the project. It includes the original project description, survey results on other projects, implementation, future work and some issues for the project. 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 and our implementation. Section 7 lists some work may be done in the future, includes some unsolved problems due to technical limitations and some possible directions of this project. Section 8 briefly describes the works related. And we put some conclutions at the end of this report.

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.

We have some challenges. Due to cross-browser communication is impossible because WebRTC[2] have not been supported by most of browsers stable versions, so intermediate data have to come back to server. This will become a bottleneck. And because the server-browser communication via HTTP, the interaction between server and clients is much slower than in clusters. Browser is also not a fast execution environment. And in the future, we can extend the functionality more by enabling other node.js[3] 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. 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.

2. 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.

3. Underscore.js[9]: 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.

4. Express: 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.

3

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   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 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	mapend
MAP_ALL_END	Server to Client	map all end
REDUCE	Server to Client	reduce
REDUCEDATA	Client to Server	reduce data
REDUCEEND	Client to Server	reduce end
COMPLETE	Server to Client	complete

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

6.2   Key-to-reducer Bucketing

6.3   Data Synchronization Problems

7   Future Works

8   Related Works

9   Conclusion

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] https://developer.mozilla.org/en/Rhino_Shell