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

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.

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[6]: 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[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.

   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[8]: 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), 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.

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.

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.

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.

5.5 API Specification

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://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