Open-Source Report

Proof of knowing your stuff in CSE312

Guidelines

Provided below is a template you must use to write your reports for your project.

Here are some things to note when working on your report, specifically about the **General Information & Licensing** section for each technology.

- **Code Repository**: Please link the code and not the documentation. If you'd like to refer to the documentation in the **Magic** section, you're more than welcome to, but we need to see the code you're referring to as well.
- **License Type**: Three letter acronym is fine.
- **License Description**: No need for the entire license here, just what separates it from the rest.
- **License Restrictions**: What can you *not* do as a result of using this technology in your project? Some licenses prevent you from using the project for commercial use, for example.

Also, feel free to extend the cell of any section if you feel you need more room.

If there's anything we can clarify, please don't hesitate to reach out! You can reach us using the methods outlined on the course website or see us during our office hours.

Koa.js

General Information & Licensing

Code Repository	https://github.com/koajs/koa/
License Type	MIT license
License Description	(The MIT License) Copyright (c) 2019 Koa contributors Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the 'Software'), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions: The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software. THE SOFTWARE IS PROVIDED 'AS IS', WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND

	NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
License Restrictions	● Liability ● Warranty

Magic ⋆★˚｡⋆.˚☽˚｡⋆⁀➷｡˚★彡⋆༄

In the previous TCP Handling report, we mentioned that when we use the `listen` method of the Koa Application Object to listen to the TCP port and start the HTTP service, the Koa will internally creates a Node.JS HTTPServer (a node built-in module). And then the HTTPServer will create a TCP server(from built-in `net` module).

The TCP Server object will listen to the `connection` event of the TCP socket, and whenever a new connection establishes. And when the TCP socket receives any data, the \`data\` event of the TCP socket will be fired and this event is listened by a function called `socketOnData` in the HTTP Server object.

Inside the \`sockerOnData\`, it asserts if the socket is pause, and then pass the received data from the TCP socket to the `execute` function of the parser.

```
https://github.com/nodejs/node/blob/main/lib/_http_server.js#L779
function socketOnData(server, socket, parser, state, d) {
   assert(!socket._paused);
   debug('SERVER socketOnData %d', d.length);

   const ret = parser.execute(d);
   onParserExecuteCommon(server, socket, parser, state, ret, d);
}
```

The \`sockerOnData\` function is simple, it asserts if the socket is pause, and then pass the received data from the TCP socket to the `execute` function of the HTTP parser in order to parse the HTTP request.

And the parser is imported from the \`_http_common\` earlier in the code.

```
const {
   parsers,
   freeParser,
   ...
} = require('_http_common');
```

In the `_http_common.js` file, we can see that the parsers is actually assigned to instance of HTTPParser when it's allocated.

```
const parsers = new FreeList('parsers', 1000, function parsersCb() {
   const parser = new HTTPParser();

   cleanParser(parser);

   parser[kOnHeaders] = parserOnHeaders;
   parser[kOnHeadersComplete] = parserOnHeadersComplete;
   parser[kOnBody] = parserOnBody;
   parser[kOnMessageComplete] = parserOnMessageComplete;

   return parser;
});
```

The `http_parser` is a library officially called `llhttp`, it detects the first few characters of the data from the socket, if the data contains the string `HTTP/` which can be used to determine if the incoming data follows the HTTP protocol,

The llhttp parser is actually a huge finite-state machine.

```
const switchType = this.load('type', {
   [TYPE.REQUEST]: n('start_req'),
   [TYPE.RESPONSE]: n('start_res'),
}, n('start_req_or_res'));
```

Parse the Control Line

Initially, the code will detect if the incoming TCP message a HTTP request or response by putting the state machine to the `'start_req_or_res'` state.

```
n('start_req_or_res')
   .peek('H', this.span.method.start(n('req_or_res_method')))
   .otherwise(this.update('type', TYPE.REQUEST, 'start_req'));
```

In this state, it reads the first byte of the message to see if the message starts with character `H`. If it starts with character `H`, then it goes to the `'req_or_res_method'` state to have further check to confirm if it's a request or response. If not, it will consider the message is a request and jumps to the `'start_req'` state because the HTTP response must start with `H` (First field of the HTTP response is the http version).

```
n('req_or_res_method')
```

```
    .select(H_METHOD_MAP, this.store('method',
        this.update('type', TYPE.REQUEST, this.span.method.end(
                                    this.invokePausable('on_method_complete',
ERROR.CB_METHOD_COMPLETE, n('req_first_space_before_url')),
        )),
    ))
    .match('HTTP/', this.span.method.end(this.update('type', TYPE.RESPONSE,
        this.span.version.start(n('res_http_major')))))
    .otherwise(p.error(ERROR.INVALID_CONSTANT, 'Invalid word encountered'));
```

In the `'req_or_res_method'` state, it firstly use `select` method to match the message text and map it to a value. The `H_METHOD_MAP` object is a map that the keys are the methods that starts with `H`. If it matches, it will store the request method code and update the `type` to `TYPE.REQUEST`. And then jumps to `on_method_complete`. Otherwise, it will try to match `HTTP/`, if the message matches, it tells that the message is a HTTP response and then it will update the `type` to `TYPE.RESPONSE` and jumps to `res_http_major` to parse the HTTP version of the response.

```
n('after_start_req')
    .select(METHOD_MAP, this.store('method', this.span.method.end(
         this.invokePausable('on_method_complete', ERROR.CB_METHOD_COMPLETE,
n('req_first_space_before_url'),
        ))))
    .otherwise(p.error(ERROR.INVALID_METHOD, 'Invalid method encountered'));
```

After it jumps to `start_req`, it will jumps to `after_start_req` which will use `select` method matches the first few characters of the messages with the `METHOD_MAP` which the keys are the methods. It will stored the matched Request Method, fire the `onMethodComplete` event and then jumps to `req_first_space_before_url` state. Otherwise, it means the method from the incoming message is invalid.

```
n('req_first_space_before_url')
    .match(' ', n('req_spaces_before_url'))
        .otherwise(p.error(ERROR.INVALID_METHOD, 'Expected space after
method'));
```

Parsing the URL

In the HTTP protocol, a whitespace will follow by the method name in the control line of the Reqeust message, and then follows by the url. So the `req_first_space_before_url` state will try to match this white space, if it found the whitespace, it will jump to the `req_spaces_before_url` state, otherwise it will throw an error indicating that it expected space after the method.

```
n('req_spaces_before_url')
    .match(' ', n('req_spaces_before_url'))
    .otherwise(this.isEqual('method', METHODS.CONNECT, {
        equal: url.entry.connect,
        notEqual: url.entry.normal,
    }));
```

Then in the `'req_spaces_before_url'` state, it first detects if the message contains extra white spaces after method name, if so, it will jump back to the begin of the `'req_spaces_before_url'` state. Otherwise, it will do a conditioning here. If the method is Connect, it will `url.entry.connect` to parse the url, if not, it will jump to `url.entry.normal` to parse the url.

```
import { URL } from './url';

…

this.url = new URL(p, mode);

…

const url = this.url.build();
```

The `url` object is the return value from `this.url.build`, `this.url` object is create from URL class, which locates in the `url.js`, and the build method locates in **Line62**.

We can see that it builds the `entry` by this, so `url.entry.normal` will make the state machine jump to `entry_normal` state.

```
const entry = {
   connect: this.node('entry_connect'),
   normal: this.node('entry_normal'),
};

entry.normal
   .otherwise(this.spanStart('url', start));
```

Then it jumps to the `start` state.

```
start
   .peek([ '/', '*' ], this.spanStart('path').skipTo(path))
   .peek(ALPHA, this.spanStart('schema', schema))
   .otherwise(p.error(ERROR.INVALID_URL, 'Unexpected start char in url'));
schema
   .match(ALPHA, schema)
   .peek(':', this.spanEnd('schema').skipTo(schemaDelim))
   .otherwise(p.error(ERROR.INVALID_URL, 'Unexpected char in url schema'));
path
   .match(this.URL_CHAR, path)
```

```
    .otherwise(this.spanEnd('path', queryOrFragment));
queryOrFragment
    .match('?', this.spanStart('query', query))
    .match('#', this.spanStart('fragment', fragment))
    .otherwise(p.error(ERROR.INVALID_URL, 'Invalid char in url path'));

query
    .match(this.URL_CHAR, query)
    // Allow extra '?' in query string
    .match('?', query)
    .peek('#', this.spanEnd('query')
        .skipTo(this.spanStart('fragment', fragment)))
    .otherwise(p.error(ERROR.INVALID_URL, 'Invalid char in url query'));

fragment
    .match(this.URL_CHAR, fragment)
    .match([ '?', '#' ], fragment)
    .otherwise(
        p.error(ERROR.INVALID_URL, 'Invalid char in url fragment start'));
```

Basically, in this part, the parser is trying to match `/` or `*` or alphabets, then the parser adds the character to the url, if it matches `?`, it adds the following characters to the query of the url, and if it matches `#`, it adds the following characters to the fragment of the url. This tears the url down into parts.

```
const onUrlCompleteHTTP = this.invokePausable(

    'on_url_complete', ERROR.CB_URL_COMPLETE, n('req_http_start'),

);



url.exit.toHTTP

    .otherwise(onUrlCompleteHTTP);
```

When the Url Parsing is done, it will jump to the `req_http_start` for further parsing.

```
n('req_http_start')
    .match('HTTP/', checkMethod(METHODS_HTTP,
        'Invalid method for HTTP/x.x request'))
    .match('RTSP/', checkMethod(METHODS_RTSP,
        'Invalid method for RTSP/x.x request'))
    .match('ICE/', checkMethod(METHODS_ICE,
        'Expected SOURCE method for ICE/x.x request'))
    .match(' ', n('req_http_start'))
    .otherwise(p.error(ERROR.INVALID_CONSTANT, 'Expected HTTP/'));
```

The parser tries to match the `HTTP/` in order to parse the HTTP version of the request. The `checkMethod` will jump to the `req_http_version` state.

Then it uses the following states, `req_http_major, req_http_dot, req_http_minor` to split the version code into major version and minor version by separator `.`

When this is done, it jumps to `req_http_end`, and then `req_http_complete`.

```
n('req_http_complete')
   .match([ '\r\n', '\n' ], n('headers_start'))
        .otherwise(p.error(ERROR.INVALID_VERSION,   'Expected   CRLF   after
version'));
```

Then it jumps to 'headers_start' after it matches CRLF after the version code.

Parse the Headers

```
n('headers_start')
   .match(' ',
       this.testLenientFlags(LENIENT_FLAGS.HEADERS, {
           1: n('header_field_start'),
       }, p.error(ERROR.UNEXPECTED_SPACE, 'Unexpected space after start
line')),
   )
   .otherwise(n('header_field_start'));

n('header_field_start')
   .match('\r', n('headers_almost_done'))
   /* they might be just sending \n instead of \r\n so this would be
    * the second \n to denote the end of headers*/
   .peek('\n', n('headers_almost_done'))
   .otherwise(span.headerField.start(n('header_field')));

n('header_field')
   .transform(p.transform.toLower())
   // Match headers that need special treatment
   .select(SPECIAL_HEADERS, this.store('header_state',
'header_field_colon'))
   .otherwise(this.resetHeaderState('header_field_general'));
```

In the 'headers_start' state, it will jump back to 'headers_start' if it detects whitespace, otherwise it jumps to 'header_field_start' to parse the field name(header key). This is used to eliminate the whitespaces in front of the actual header field.

Then if it detects `\r\n` in the beginning of a new line, that means this is an empty line, which means the header part of the request is done, so it will jump to the `headers_almost_done` state. Otherwise, it jumps to 'header_field' state to parse the header.

In 'header_field', it initially transform the line to lowercase, and try to match special headers such as `connection`, `content-length` and so on, their header state code will be recorded into `header_state`, and then jumps to `header_field_colon`, otherwise, the `header_state` code will be set to `HEADER_STATE.GENERAL` and then jumps to 'header_field_general'.

In the 'header_field_general' state, if the parser encounters illegal chaaracters(TOKENS), the parser will jump back to 'header_field_general' state and ignore the illegal characters.

When it matches `:`, it means the header field ends, therefore it jups to `on_header_field_complete` state.

```
const onHeaderFieldComplete = this.invokePausable(
              'on_header_field_complete',     ERROR.CB_HEADER_FIELD_COMPLETE,
n('header_value_discard_ws'),
);
…
n('header_field_general_otherwise')
    .peek(':', span.headerField.end().skipTo(onHeaderFieldComplete))
    .otherwise(p.error(ERROR.INVALID_HEADER_TOKEN, 'Invalid header token'));
```

In 'header_value_discard_ws' state

```
n('header_value_discard_ws')
    .match([ ' ', '\t' ], n('header_value_discard_ws'))
    .match('\r', n('header_value_discard_ws_almost_done'))
    .match('\n', this.testLenientFlags(LENIENT_FLAGS.HEADERS, {
         1: n('header_value_discard_lws'),
    }, p.error(ERROR.INVALID_HEADER_TOKEN, 'Invalid header value char')))
    .otherwise(span.headerValue.start(n('header_value_start')));
```
(llhttp/src/llhttp/http.ts#L551)

If the parser matches whitespace ot `\t`, it will jump back to 'header_value_discard_ws' state to ignore the whitespace. If it matches `\r`, it means the header value is about to end, then it will jump to the 'header_value_discard_ws_almost_done' state . If it matches `\n`, it means the header value is also about to end, then it will jump to the 'header_value_discard_lws' state. Otherwise, it means the value is continuing, so it will jump to 'header_value_start' state.

```
n('header_value_start')
    .otherwise(this.load('header_state', {
         [HEADER_STATE.UPGRADE]: this.setFlag(FLAGS.UPGRADE, fallback),
         [HEADER_STATE.TRANSFER_ENCODING]: this.testFlags(
              FLAGS.CHUNKED,
              {
                                                                            1:
forbidAfterChunkedInRequest(this.setFlag(FLAGS.TRANSFER_ENCODING,
toTransferEncoding)),
              },
              this.setFlag(FLAGS.TRANSFER_ENCODING, toTransferEncoding)),
   [HEADER_STATE.CONTENT_LENGTH]: n('header_value_content_length_once'),
         [HEADER_STATE.CONNECTION]: n('header_value_connection'),
    }, 'header_value'));
```
(llhttp/src/llhttp/http.ts#L603)

In this state, it jumps to several different state to process different special headers, such as the `content-length` header (it's special because the parser needs its information to parse the body). Then it jumps to `'header_value'`.

```
n('header_value')
   .match(HEADER_CHARS, n('header_value'))
   .otherwise(n('header_value_otherwise'));
```

In `'header_value'` state, it tries to match the characters in ``HEADER_CHARS`` list, otherwise it will jump to `'header_value_otherwise'` state because the line is terminating.

```
n('header_value_otherwise').peek('\r',
span.headerValue.end().skipTo(n('header_value_almost_done')))
   .otherwise(checkLenient);

n('header_value_almost_done')
   .match('\n', n('header_value_lws'))
   .otherwise(p.error(ERROR.LF_EXPECTED,
       'Missing expected LF after header value'));
```

In `'header_value_otherwise'` and `'header_value_almost_done'` state, it tries to match `\r` and `\n` which indicates the header line terminates.

Then it jumps to `'header_value_lws'` state,

```
n('header_value_lws')
   .peek([ ' ', '\t' ],
       this.load('header_state', {
           [HEADER_STATE.TRANSFER_ENCODING_CHUNKED]:

this.resetHeaderState(span.headerValue.start(n('header_value_start'))),
       }, span.headerValue.start(n('header_value_start'))))
   .otherwise(this.setHeaderFlags(onHeaderValueComplete));
```

Here ist calls ``onHeaderValueComplete`` and then it will goes to ``header_field_start`` to parse the next header.

When the parser detects an empty line in the header field, it will jumps to ``headers_almost_done`` state, then it fires the ``onHeadersComplete`` event.

After `onHeadersComplete` event got fire, it will jump to `headers_done` state, then it begins to parse the body. It will read `content-length` bytes following the headers end, and store the data into the body.

And so far, the http request has been parsed.

```
https://github.com/nodejs/node/blob/main/lib/_http_server.js#L779
function socketOnData(server, socket, parser, state, d) {
   assert(!socket._paused);
   debug('SERVER socketOnData %d', d.length);

   const ret = parser.execute(d);
   onParserExecuteCommon(server, socket, parser, state, ret, d);
}
```

Back to the socketOnData in the http_server file, the result from the HTTPParser is stored in the `ret` variable, and then `onParserExecuteCommon` get called.

Then, the request object is built, and the `parserOnIncoming` get called, and it will build a nodejs `response` object for the http handling function to build http response (The process afterward will be described back in the TCP report since the parsing process is done).